12-2014

Fault Tolerance in Distributed Systems Using Self-Stabilization

Yihua Ding
Clemson University

Fault Tolerance in Distributed Systems Using Self-Stabilization

---

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

---

by
Yihua Ding
December 2014

Accepted by:
James Wang, Committee Chair
Pradip Srimani, Committee Co-Chair
Jason Hallstrom
Feng Luo

Abstract

Self-stabilization is an optimistic paradigm to provide autonomous resilience against an unlimited number of transient faults in distributed systems. A self-stabilizing system guarantees an eventual return to a legitimate operating state beginning with an unknown initial state, including a state that arises as the result of an unanticipated transient fault (e.g., node failure, state corruption, or node mobility in mobile systems). This thesis focuses on the fault tolerance in distributed systems using self-stabilization, and presents a collection of self-stabilizing algorithms for well-known problems in distributed systems.

Two variants of dominating set (i.e., weakly connected dominating set and positive influence dominating set) have been widely used in distributed system applications, such as network security, facility location, message routing and others. In this thesis, three new self-stabilizing algorithms are proposed to compute a minimal weakly connected dominating set in the arbitrary network graph under different models; they outperform the best possible solution existing in the literature. Recently, the concept of a positive influence dominating set has been used for the selection of influential users in the social network. However, this solution does not consider that (1) the mutual influence between two neighboring nodes is in general asymmetric; and (2) each node has different tolerance level (sensitivity) towards neighbor's influence. To overcome these drawbacks, I propose to select the users in the minimal weighted positive influence dominating set of a social network graph as the influential users, and then present the first polynomial time self-stabilizing algorithm to compute a minimal weighted positive influence dominating set in an arbitrary social network.

In a traditional self-stabilizing algorithm, the desired global property (the relevant service in the system) is not guaranteed during the convergence interval; thus the concept of safe convergence was introduced to minimize possible inconvenience. A self-stabilizing algorithm has the safe convergence property if it first converges to a safe (sub-optimal) state in constant time, and then converges

to a legitimate (optimal) state without breaking safety in the process. Safe convergence property is especially attractive as it provides a measure of safety (desired service at a sub-optimal level) during the convergence interval until the optimal legitimate state is reached. This thesis presents the first set of self-stabilizing algorithms with safe convergence property for packing and alliance problems in arbitrary network graphs.

Dedication

To my family.

Acknowledgments

First of all, I would like to thank my advisors, Dr. James Wang and Dr. Pradip Srimani, for their support, guidance and encouragement on my way towards the Ph.D. degree.

Second, I would like to thank Dr. Feng Luo and Dr. Jason Hallstrom for serving on my committee and giving me their suggestions on my Ph.D. research.

Third, I would like to thank Liang Dong, Lin Li, and Xuebo Song for being great group mates. Special thanks belong to my colleague, friend, and wife, Yuanyuan Zhang, for her help and moral support.

Last but not least, I would like to thank my parents, Deyu Ding and Suqin Liu, for always believing in me and supporting me no matter the cost.

Table of Contents

List of Tables

List of Figures

Chapter 1

Fault-Tolerant Distributed Algorithms and Self-Stabilization

1.1 Fault Tolerance

In the past couple of decades, we have observed a tremendous growth in demand for highly available and reliable (fault-tolerant) distributed systems that provide desirable services automatically without initialization and any external intervention after faults occur. In this context, various fault-tolerant algorithms have been proposed in order to increase system stability and avoid disasters, such as Amazons S3 and Gmail incidents [23, 1], triggered by faults.

The faults can be classified as *persistent* and *transient* based on their duration. A persistent fault remains in the distributed system all the time, while a transient fault exists for a limited period of time. The fault tolerance approaches can be classified as *masking* and *non-masking*. A masking fault tolerance approach aims at masking the effects of the faults using redundancy (additional hardware or software). Such an approach is able to make the system service always available, but the redundancy may considerably increase the cost of the system. The non-masking fault tolerance approach is relatively cheap, but the drawback of this approach is that it accepts the temporary unavailability of the system service for a limited time.

This thesis is interested in the transient faults, such as topological change, memory corruption. The proposed algorithms (classified as non-masking fault tolerance approaches) are able

to recover the system from transient faults in finite time, as long as no transient faults occur long enough time and the program code is intact.

1.2 Self-Stabilization

Self-stabilization, a non-masking fault tolerance approach, is an optimistic paradigm to provide autonomous resilience against an unlimited number of transient faults in the distributed systems. An algorithm is self-stabilizing iff it reaches some legitimate state, starting from an arbitrary state [8], and remains legitimate thereafter if no transient faults occur. Without initialization and any external intervention, a self-stabilizing system can recover from an unlimited number of transient faults in a reasonable time based only on the local knowledge.

1.2.1 System Model

In a self-stabilizing algorithm, each node maintains a set of local variables that defines the *local state* of the node. The union of the local states of all nodes in the system is called the *global state*. A node can access the local states of its immediate neighbors (communication model will be discussed next). However, a node can write to only its own local variables. A global state can be either *legitimate* or *illegitimate*. A legitimate state is a global state where the desired global property (hence, the relevant service in the system) is guaranteed, while an illegitimate state is a global state where the desired service is not available due to the transient faults. A system transits through the global states, which conceptually forms a directed graph where each vertex represents a global state and each edge represents the transition between two different global states (called *state transition graph*). A self-stabilizing algorithm is said to be *silent* if it terminates, i.e., it is guaranteed to reach a vertex of the state transition graph without any outgoing edges. The purpose of designing a self-stabilizing algorithm is to make the system either terminate at a vertex representing a legitimate state, or transit through the vertices representing legitimate states in finite time.

1.2.2 Communication Model

Most self-stabilizing algorithms employ the following three models to achieve information transmission between nodes in the distributed system.

- *State-reading model* (or *shared-memory model with composite atomicity*): A node can read all variables stored at its immediate neighbors, but can change only its own variables. This model considers reading all variables stored at immediate neighbors and updating its own local variables as an atomic action. It is the most common model used in designing self-stabilizing algorithms.

- *Read/write atomicity model*: Similar to state-reading model, this model also assumes that a node can read all variables stored at its immediate neighbors, but can change only its own variables. It is to be noted that this model considers an atomic action as either a single read operation or a single write operation (but not both).

- *Message-passing model*: In this model, the communication between neighboring nodes is achieved by sending and receiving messages (information is exchanged in the form of messages). An atomic action contains either sending a message to one of its neighbors or receiving such a message from a single neighbor (but not both). This model is more complicated than state-reading model and read/write atomicity model due to communication delay and message corruption.

The state-reading model and read/write atomicity model are usually implemented by message-passing. All algorithms discussed in this thesis use state-reading model as is customary in the most self-stabilizing algorithms.

1.2.3 Algorithm Design

To design a self-stabilizing algorithm, we need to define some predicate(s) at each node that depends on the local state at the node and those at the neighboring nodes (since typically only the states of the immediate neighbors are available to a node); each node can then determine if it is locally legitimate (with respect to the given global state of the problem at hand) from the knowledge of its own state and the states of the adjacent nodes. Most of the self-stabilizing algorithms follow this approach.

A self-stabilizing algorithm is usually written as a set of rules at each node. Each rule at a node consists of a *condition* and an *action*:

if *condition* **then** *action*

A *condition* is a boolean predicate involving the states of the node and its neighbors. If one or more conditions on a node are satisfied, i.e., some boolean predicates are true, the node is *privileged*. If a privileged node takes an *action* (also called a *move*), it changes its own local variables.

The self-stabilizing algorithm can be either *uniform* or *semi-uniform*. An algorithm is uniform if all nodes use the same program, and is semi-uniform if there exists some "special" node(s) that behaves differently from the rest of the nodes.

1.2.4 Runtime Schedulers

In any global state, there may exist multiple privileged nodes. A runtime scheduler (also called a *daemon*) is assumed to select the privileged node(s) such that only selected privileged node(s) is able to make moves. If a privileged node is selected by the scheduler, it makes a move by changing its local variables. The scheduler is a virtual entity (i.e., no physical existence of a scheduler). In general, the scheduler plays a role of both scheduler and adversary, which is important to the analysis of the correctness of the algorithm and its worst-case time complexity. Specifically, it ensures the progress of the system to verify the correctness of the algorithm by performing as a scheduler, and maximizes the execution sequence to explore the worst-case scenario by performing as an adversary.

The most common schedulers include:

- *Central scheduler* (or *serial scheduler*): In any global state, the central scheduler selects exactly one privileged node to move.
- *Synchronous scheduler*: In any global state, the synchronous scheduler selects all the privileged nodes to move.
- *Distributed scheduler*: In any global state, the distributed scheduler selects a non-empty subset of the privileged nodes to move.

It is to be noted that any self-stabilizing algorithm using a distributed scheduler also works for a central scheduler or a synchronous scheduler. It is easier to develop and analyze the algorithms using central scheduler than using synchronous or distributed scheduler, but the synchronous and distributed schedulers are more desirable as they are closer to the real life distributed system. Goddard and Srimani [27] showed that any self-stabilizing algorithm using a central scheduler can be converted to one that self-stabilizes (1) using a synchronous scheduler with a constant slowdown

factor, and (2) using a distributed scheduler with a linear slowdown factor in terms of the number of nodes in the distributed system.

A scheduler is called *fair* if any privileged node is guaranteed to be selected by the scheduler after a finite number of steps or to become unprivileged because of the moves of its neighbors. Otherwise, the scheduler is called *unfair*. [44] showed that any self-stabilizing algorithm using a fair central scheduler can be converted to one that self-stabilizes using an unfair central scheduler with a slowdown factor of $O(n^3)$ steps, where n is the number of nodes in the distributed system.

1.2.5 Complexity Measures

The complexity measures used to evaluate a self-stabilizing algorithm include time complexity, space complexity and message complexity. All self-stabilizing algorithms discussed in this thesis use state-reading model, thus only time and space complexities are considered. The time complexity of a self-stabilizing algorithm can be measured in terms of either the maximum number of *steps* or the maximum number of *rounds* needed for an algorithm to reach a legitimate state, starting from an arbitrary illegitimate state.

An algorithm executes one step if (1) a unique privileged node makes a move under a central scheduler, or (2) all privileged nodes make moves simultaneously under a synchronous scheduler, or (3) a non-empty subset of privileged nodes make moves simultaneously under a distributed scheduler. Round has different definitions in the literature. Mostly, a round is defined as the minimal prefix of an execution sequence of steps where each continuously privileged node is selected by the scheduler to take actions. However, sometimes a round is defined as the minimal prefix of an execution sequence of steps where each node privileged in the initial state is either chosen by the scheduler to take actions or becomes unprivileged due to the change of state(s) of its neighbor(s); it is possible in many cases that one round will consist of a large number of steps. For self-stabilizing algorithms using synchronous schedulers, a round is equivalent to a step. All proposed self-stabilizing algorithms in this thesis are evaluated in terms of steps.

1.2.6 Local Knowledge

A node is able to access a partial view of the global state, which depends on the connectivity of the system and computation model. Most self-stabilizing algorithms use *distance-1 model*: a node

can access the local states of its immediate neighbors, and hence the condition of each rule involves the local states of the node and those of its immediate neighbors. An extension of this model is *distance-k model*, where a node can access the local states of nodes up to distance-k, and hence the condition of each rule involves the local states of the node and those of the nearby nodes within distance-k [31]. There are certain algorithmic problems that can be solved more easily on an extended model [21, 31, 47].

As shown in [47, 31], any self-stabilizing algorithm using the distance-2 model and an unfair central scheduler can be converted to a distance-1 model algorithm using an unfair distributed scheduler at an increased $O(n^2)$ computational cost, where n is the number of nodes in the distributed system.

1.2.7 Safe Convergence

Recently, a new concept of *safe convergence* has been introduced in [32] in the context of self stabilizing graph algorithms. In a traditional self-stabilizing algorithm, the desired global property (hence, the relevant service in the system) is not guaranteed during the convergence interval; the concept of safe convergence was introduced to limit this inconvenience to a minimum possible. A self-stabilizing algorithm has the safe convergence property iff it first converges to a safe (feasible but sub-optimal) state in constant time, and then converges to a legitimate (optimal) state without breaking safety in the process. Safe convergence property is especially attractive since it provides a measure of safety (desired service at a sub-optimal level) during the convergence interval until the optimal legitimate state is reached. Several self-stabilizing algorithms with safe convergence property have been proposed for graph theoretical problems, such as minimal independent dominating set, connected dominating set and others [32, 34, 36]; all of these algorithms have assumed a synchronous scheduler and unique node IDs; these algorithms reach a corresponding safe state in constant number of steps starting from an arbitrary initial state and subsequently converge to a corresponding legitimate state without breaking safety in the process.

Definition 1.2.1 *A self stabilizing algorithm is said to have* safe convergence *property when it quickly converges to a safe state which is not necessarily optimal and then converges to a legitimate state without violating safety during the convergence interval.*

1.3 Self-Stabilizing Algorithms for Graph Theoretic Invariants

Most essential services in large scale networked distributed systems (ad hoc, wireless or sensor) involve maintaining a global predicate over the entire network. Each node has limited computing and communication capabilities, limited storage, and limited energy resource. The network needs to achieve a larger global task (defined by some invariance relation on the global state of the network). The participating sensor nodes can no longer keep track of even a small fraction of the knowledge about the global network due to limited storage. Self-stabilization is an optimistic paradigm to provide scalability in coordinating the nodes and implementing fault tolerance in such networks. This section first introduces the graph model used in this thesis, and then provides a survey of self-stabilizing algorithms for classical graph theoretic invariants, e.g., dominating set and its variants, graph packing, and alliances. There already exist self-stabilizing algorithms presented in the literature for some of these problems; this thesis reconsiders them to design new algorithms with safe convergence property or to reduce complexities (including time and space complexities).

1.3.1 Graph Model

A network or a distributed system is modeled by an undirected graph $G = (V, E)$, where V is the set of nodes (processors), $|V| = n$, and E is the set of edges (connections or links), $|E| = m$. For a node i, $N(i)$, its *open neighborhood*, denotes the set of nodes adjacent to node i; $N[i] = N(i) \cup i$ denotes its *closed neighborhood*. For a node i, $N^2(i) = \cup_{j \in N[i]} N(j) - \{i\}$, its *2-hop open neighborhood*, denotes the set of nodes that are at most distance of 2 from node i. Each node $j \in N(i)$ is called a *neighbor* of node i and each node $j \in N^2(i)$ is called a *2-neighbor* of node i. The distance $dist(i, j)$ is the number of edge(s) in the shortest path between nodes i and j. The diameter of G is denoted to be Δ, $\Delta = \max_{i,j \in V}\{dist(i, j)\}$. An example graph is shown in Figure 1.1, where $n = m = 5$. $N(1) = \{4, 5\}$, $N[1] = \{1, 4, 5\}$, $N^2(2) = \{1, 4, 5\}$, $dist(2, 3) = 3$, $\Delta = 3$.

1.3.2 Dominating Set and Its Variants

Definition 1.3.1 *In an arbitrary graph $G = (V, E)$, a set $\mathcal{S} \subseteq V$ is a* dominating *set if each node $i \in \{V - \mathcal{S}\}$ is adjacent to at least one node in $\mathcal{S}$, i.e., $N(i) \cap \mathcal{S} \neq \emptyset$. A dominating set $\mathcal{S}$ is called a*

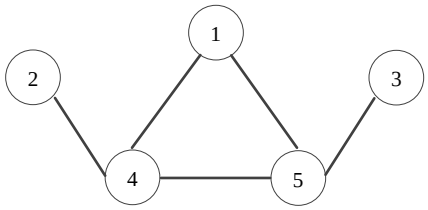

Figure 1.1: An example graph with 5 nodes

minimal dominating set *iff there does not exist a node* $i \in \mathcal{S}$ *such that* $\mathcal{S} - \{i\}$ *is a dominating set.*

Minimal dominating set is an important communication structure in the distributed systems. If resources are put on the nodes of a minimal dominating set, then each node in the system is able to access the resources in at most one hop. The first self-stabilizing algorithm for minimal dominating set is introduced in [30]; the algorithm stabilizes in $O(n^2)$ steps under a central scheduler, where n is the number of nodes in the network. In [52], a synchronous minimal dominating set algorithm is proposed. This algorithm stabilizes in $O(n)$ steps under a synchronous scheduler. Turau [46] presents the first minimal dominating set algorithm under a distributed scheduler. It stabilizes in at most $9n$ steps. Subsequently, [26] presents a new algorithm for the minimal dominating set problem that stabilizes in at most $5n$ steps under a distributed scheduler. All of the above algorithms require $O(\log n)$ bits at each node. They all assume that each node has a globally unique identifier. [14] proposes a new self-stabilizing algorithm for minimal dominating set; it has safe convergence property under a synchronous scheduler in the sense that starting from an arbitrary state, it quickly converges to a dominating set (a safe state) in two steps, and then stabilizes in a minimal dominating set (the legitimate state) in $O(n)$ steps without breaking safety during the convergence interval, where n is the number of nodes. Space requirement at each node is $O(\log n)$ bits.

Definition 1.3.2 *In an arbitrary graph* $G = (V, E)$*, a set* $\mathcal{S} \subseteq V$ *is a* total dominating set *if each node* $i \in V$ *is adjacent to at least one node in* $\mathcal{S}$*. A total dominating set* $\mathcal{S}$ *is* minimal *iff there does not exist a node* $i \in \mathcal{S}$ *such that the set* $\mathcal{S} - \{i\}$ *is a total dominating set.*

If a minimal dominating set in a communication network represents a minimal set of servers necessary to provide an acceptable level of service (e.g., storage service), then a total dominating set represents a similar set of servers with the added capability that each server is adjacent to at least one other server. In this way, each server has a backup resource. If a node's capability as a server is

compromised in anyway, it can obtain backup from another server in the network with a minimum possible delay. Thus total dominating sets are more fault tolerant than dominating sets.

The authors in [24] present a self-stabilizing algorithm for constructing minimal total dominating sets in an arbitrary network graph; the algorithm uses distinct node IDs, assumes a central scheduler, and requires $O(\log n)$ bits of space at each node. The stabilization time of algorithm in [24] is shown in [25] to be exponential in n, where n is the number of nodes in the network. [12] presents a new self-stabilizing algorithm for minimal total dominating sets. The stabilization time is $O(n^3)$ steps using a central scheduler, and the space requirement at each node is $O(\log n)$ bits for an arbitrary graph with n nodes. This is a significant improvement over the most recent self-stabilizing algorithm for minimal total dominating set that uses $O(2^n)$ steps with the same space complexity [24, 25].

A connected dominating set (CDS) of a given graph is a dominating set whose induced subgraph is connected; it has many applications in communications, e.g., message routing and clustering, due to its domination and connectivity properties [6]. No self-stabilizing algorithm is available in the literature for minimal CDS [28], while there exist self-stabilizing algorithms to compute approximate minimum CDS for a class of special graphs [35, 36]. Since the size of a minimal CDS can be relatively very large in many network graphs, a variant of minimal CDS, called minimal weakly connected dominating set (MWCDS) has been introduced and found useful in networks [6]. A weakly connected dominating set is a dominating set where the induced subgraph of the closed neighborhood of the set is connected.

Definition 1.3.3 *In an arbitrary graph $G = (V, E)$, a set $\mathcal{S} \subseteq V$ is a* weakly connected dominating *set if it is a dominating set and the subgraph $(N[\mathcal{S}], E \cap (\mathcal{S} \times N[\mathcal{S}]))$ is connected, where $N[\mathcal{S}] = \cup_{i \in \mathcal{S}} N[i]$. A weakly connected dominating set $\mathcal{S}$ is called* minimal *iff there does not exist a node $i \in \mathcal{S}$ such that $\mathcal{S} - \{i\}$ is a weakly connected dominating set.*

The first reported self-stabilizing algorithm for MWCDS in any connected graph is proposed in [44, 53]. It assumes a distinguished node (all other nodes are identical and anonymous) and stabilizes in $O(2^n)$ steps using an unfair central scheduler, where n is the number of nodes in the network graph. Kamei and Kakugawa present an approximated self-stabilizing algorithm for minimum weakly connected dominating set in connected unit disk graphs [33] (not minimal), which assumes nodes with unique identifiers and stabilizes in $O(n^2)$ rounds using a synchronous scheduler

(a round in [33] is defined as a period from the beginning to the end of an execution of a loop of the algorithm). Subsequently, Turau and Hauck [48] propose two other self-stabilizing algorithms, which assume a distinguished node as root and nodes with unique identifiers, to construct MWCDS in any connected graph. Both algorithms stabilize in $O(nmA)$ steps under an unfair central scheduler and an unfair distributed scheduler respectively, where m is the number of edges in the graph and A is the number of moves to construct a breadth-first spanning tree. To the best of our knowledge, the best reported self-stabilizing algorithm for building a breadth-first spanning tree satisfying the model in [48] is proposed in [44], which stabilizes in $A = O(n^3)$ steps using an unfair central scheduler, and can be converted into one that self-stabilizes in $A = O(n^4)$ steps using an unfair distributed scheduler as shown in [27].

1.3.3 Graph Alliances

Definition 1.3.4 *In an arbitrary graph $G = (V, E)$, a set $\mathcal{S} \subseteq V$ is a* global offensive alliance *if each node $i \in \{V - \mathcal{S}\}$ has $|N[i] \cap \mathcal{S}| \geq |N[i] - \mathcal{S}|$. A global offensive alliance $\mathcal{S}$ is called* minimal *iff there does not exist a node $i \in \mathcal{S}$ such that the set $\mathcal{S} - \{i\}$ is a global offensive alliance.*

The authors in [43] present a self-stabilizing algorithm for constructing minimal global offensive alliance in an arbitrary network graph; the algorithm assumes a central scheduler, and uses 1 bit of space at each node. The stabilization time is $O(n^2)$ steps, where n is the number of nodes in the network. [15] proposes a new self-stabilizing algorithm for minimal global offensive alliance. It has safe convergence property under synchronous scheduler in the sense that starting from an arbitrary state, it quickly converges to a global offensive alliance (a safe state) in two steps, and then stabilizes in a minimal global offensive alliance (the legitimate state) in $O(n)$ steps without breaking safety during the convergence interval, where n is the number of nodes. Space requirement at each node is $O(\log n)$ bits.

Definition 1.3.5 *In an arbitrary graph $G = (V, E)$, a set $\mathcal{S} \subseteq V$ is a global powerful alliance if for each node $i \in V$, $|N[i] \cap \mathcal{S}| \geq |N[i] - \mathcal{S}|$. A global powerful alliance $\mathcal{S}$ is called* minimal *iff there does not exist a node $i \in \mathcal{S}$ such that the set $\mathcal{S} - \{i\}$ is a global powerful alliance.*

The authors in [54] present a self-stabilizing algorithm for constructing minimal global powerful alliance in an arbitrary network graph; the algorithm is based on the distance-two model,

where each node can read all states of nodes that are within distance two. It stabilizes in $O(n)$ steps for any graph with n nodes under a central scheduler. This algorithm can be transformed into an algorithm based on distance-one model using the distributed scheduler with a slowdown factor of $O(n^2)$ steps by employing the transformer given in [47]. The transformer requires that each node has a unique ID. Thus, the resulting algorithm needs distinct node IDs and $O(\log n)$ bits of space at each node. The stabilization time is $O(n^3)$ steps. [9] presents the first self-stabilizing minimal global powerful alliance algorithm with safe convergence. It is assumed to face a synchronous scheduler. Starting in any arbitrary state, it quickly converges to a global powerful alliance (a safe state) in 3 steps, and then stabilizes in a minimal global powerful alliance (the legitimate state) in $O(n)$ steps without breaking safety during the intermediate state transitions. The behaviors of the nodes are managed by a synchronous scheduler where all nodes privileged by the algorithm execute actions simultaneously in a step.

Definition 1.3.6 *In an arbitrary graph $G = (V, E)$, consider two functions f, g that map each node to two non-negative integers and each node $i \in V$ has a degree $\delta_i \geq g_i$. A set $\mathcal{S} \subseteq V$ is an* (f, g)-alliance *if each node $i \in \{V - \mathcal{S}\}$ has $|N(i) \cap \mathcal{S}| \geq f_i$ and each node $i \in \mathcal{S}$ has $|N(i) \cap \mathcal{S}| \geq g_i$.*

The concept of (f, g)-alliance in a graph has been first introduced and studied in [19]. Note that an (f, g)-alliance always exist for a graph when the constraints on the node degrees and the mapping functions f, g are met. An (f, g)-alliance $\mathcal{S}$ with $f_i \geq g_i$ and $\delta_i \geq g_i$ at each node i, is called *minimal* iff there does not exist a node $i \in \mathcal{S}$ such that the set $\mathcal{S} - \{i\}$ is an (f, g)-alliance. Authors in [19] have observed that the concept of (f, g)-alliance actually generalizes some already existing graph theoretical invariants; specifically, a (minimal) (f, g)-alliance $\mathcal{S}$ is (1) a (minimal) dominating set iff $f_i = 1$ and $g_i = 0$ for each $i \in V$; (2) a (minimal) k-dominating set iff $f_i = k$ and $g_i = 0$ for each $i \in V$; (3) a (minimal) total dominating set iff $f_i = g_i = 1$ for each $i \in V$; and (4) a (minimal) offensive alliance iff $f_i = \lceil \delta_i/2 \rceil$ and $g_i = 0$ for each $i \in V$. They also proposed two synchronous message-passing distributed algorithms to compute such minimal (f, g)-alliance $\mathcal{S}$: one is ID-based linear-time deterministic algorithm, and the other one is an anonymous randomized algorithm; these two algorithms are not self-stabilizing.

Recently, authors in [3] proposed a self-stabilizing algorithm for minimal (f, g)-alliance of a graph with $f_i \geq g_i$ and $\delta_i \geq g_i$ at each node i. They assume an unfair distributed scheduler and a unique ID for each node and show that the algorithm stabilizes in a legitimate state in $O(\Delta^3 n)$

steps, where Δ is the maximum node degree in the graph and n is the number of nodes. Then, they defined a *round* as the minimal prefix of an execution sequence of steps where each node privileged in the initial state is either chosen by the scheduler to take actions or becomes unprivileged due to the change of state of its neighbor(s); and showed that the algorithm enters an (f, g)-alliance in at most 4 such rounds, and then terminates in a minimal one in at most $5n+4$ additional rounds. But, it is possible in many cases that the initial round will consist of a large number of steps made by the unfair distributed scheduler; indeed, such a round can consist of $\Omega(n^2)$ steps of the unfair scheduler.

1.3.4 Graph Packing

Definition 1.3.7 *In an arbitrary graph $G = (V, E)$, a set $\mathcal{S} \subseteq V$ is a* k-packing *if $\forall i \in V$: $|N^{k-1}[i] \cap \mathcal{S}| \leq 1$. A k-packing is* maximal *if no proper superset of $\mathcal{S}$ is a k-packing. It is to be noted that $\mathcal{S}$ is a k-packing iff $\forall i, j \in \mathcal{S} : dist(i, j) \geq k+1$; $\mathcal{S}$ is a maximal k-packing iff $\forall i \in \{V - \mathcal{S}\}$ $\exists j \in \mathcal{S} : dist(i, j) \leq k$.*

The concept of k-packing, $k \geq 2$, has been used in various applications like network security, facilities location and others [22]. Authors in [20, 21] designed the first two self-stabilizing algorithms for maximal 2-packing, that stabilize in exponential time and $O(n^3)$ time respectively using a central scheduler, where n is the number of nodes in the graph. Subsequently, other self-stabilizing algorithms have appeared [25, 39, 41]. The most recent self-stabilizing algorithm for maximal 2-packing is given in [42] that stabilizes in $O(n^2)$ synchronous steps (using a synchronous scheduler). All of the above algorithms require $O(\log n)$ bits at each node and assume that each node has a unique ID; none of them enjoys safe convergence property. Self-stabilizing k-packing algorithms are presented in [25, 39]; both algorithms stabilize in exponential time under a central scheduler.

1.4 Contributions

This thesis focuses on the fault tolerance using self-stabilization, and presents a collection of self-stabilizing algorithms for well-known problems in distributed systems. The self-stabilizing algorithms discussed in this thesis can be easily used as building blocks to develop fault-tolerant distributed systems. Specifically, the following results are presented in this thesis [1]:

[1] This thesis is based on my work in [10, 11, 13, 15, 16].

In Chapter 2, three new self-stabilizing algorithms (`MWCDS-S`, `MWCDS-C` and `MWCDS-D`) are presented to compute minimal weakly connected dominating set (one of the most widely used variants of the dominating set) for an arbitrary network graph. They all assume the existence of a distinguished root node and have $O(\log n)$ bits of space requirement at each node. Algorithm `MWCDS-S` terminates in $O(n)$ steps using a synchronous scheduler and unique node IDs. Algorithm `MWCDS-C` stabilizes in $O(n^4)$ steps using an unfair central scheduler; it assumes all non-root nodes are identical and anonymous. Algorithm `MWCDS-D` stabilizes using an unfair distributed scheduler with the same time and space complexities as algorithm `MWCDS-C`, but it assumes unique node IDs. This represents a significant improvement over the best possible solution existing in the literature.

In Chapter 3, an application of positive influence dominating set (another variant of the dominating set) in the social network, i.e., influential users selection, is studied. Then, I propose to select the users in the minimal weighted positive influence dominating set of a social network graph as the influential users. The usage of minimal weighted positive influence dominating set overcomes the drawbacks of the positive influence dominating set approach, i.e., regardless of the asymmetric influence between two neighboring users and different sensitivities of users to influence. At last, a new self-stabilizing algorithm to compute a minimal weighted positive influence dominating set for an arbitrary social network is proposed.

In Chapter 4, the first set of self-stabilizing algorithms with safe convergence property are proposed for packing and alliance problems in arbitrary network graphs. Starting from an arbitrary state, the proposed algorithms first converge to a safe (feasible but sub-optimal) state in constant time, and then converge to a legitimate (optimal) state in $O(n)$ steps without breaking safety during the intermediate state transitions, where n is the number of nodes in the network graph.

In Chapter 5, simulations are conducted for the proposed algorithms to evaluate their correctness and efficiencies.

Chapter 2

Self-Stabilizing Algorithms for Minimal Weakly Connected Dominating Set

2.1 Model and Terminology

Consider an arbitrary connected graph $G = (V, E)$, a set $\mathcal{S} \subseteq V$ is a *weakly connected dominating* set if it is a dominating set and the subgraph $(N[\mathcal{S}], E \cap (\mathcal{S} \times N[\mathcal{S}]))$ is connected, where $N[\mathcal{S}] = \cup_{i \in \mathcal{S}} N[i]$. A weakly connected dominating set $\mathcal{S}$ is called *minimal* iff there does not exist a node $i \in \mathcal{S}$ such that $\mathcal{S} - \{i\}$ is a weakly connected dominating set.

Minimal weakly connected dominating set (MWCDS) has many applications in communications, e.g., message routing and clustering, due to its domination and connectivity properties [6]. A survey of self-stabilizing algorithms for constructing an MWCDS is given in Section 1.3.

In this chapter, three new self-stabilizing algorithms, `MWCDS-S`, `MWCDS-C` and `MWCDS-D`, are proposed for minimal weakly connected dominating sets in an arbitrary connected graph. For an arbitrary connected graph with n nodes:

- Algorithm `MWCDS-S` terminates in $O(n)$ steps using a synchronous scheduler; it uses a distinguished root node and assumes unique node IDs. The space requirement at each node is

$O(\log n)$ bits.

- Algorithm `MWCDS-C` stabilizes in $O(n^4)$ steps using an unfair central scheduler; it uses a distinguished root node while other nodes are identical and anonymous. The space requirement at each node is $O(\log n)$ bits.

- Algorithm `MWCDS-D` stabilizes in $O(n^4)$ steps using an unfair distributed scheduler; it uses a distinguished root node and assumes unique node IDs. The space requirement at each node is $O(\log n)$ bits.

Execution Model: The proposed algorithms assume the existence of a distinguished node called *root*; this assumption is especially suitable for sensor network or ad hoc networks based on cluster architecture, where a base station or a cluster head has more memory, and is more powerful than other ordinary nodes [7]. The execution of each proposed algorithm at a node is managed by a runtime scheduler. A node is privileged in a given system state iff it is enabled to move by the rule of the algorithm. The algorithms use state-reading model (as opposed to read/write atomicity model [18]); each node knows only its own state and the local states of its immediate neighbors (distance-one model) as is customary in almost all self-stabilizing algorithms.

Definition 2.1.1 $\boldsymbol{\mu_i}$ *is used to denote the distance of any node* i *to the distinguished root node (i.e.,* $\mu_i = dist(i, root)$*).*

2.2 Algorithm `MWCDS-S`

In this section, a new self-stabilizing algorithm (called algorithm `MWCDS-S`) is proposed to compute minimal weakly connected dominating set in an arbitrary connected network graph. For an arbitrary connected graph with n nodes, algorithm `MWCDS-S` terminates in $O(n)$ steps under a synchronous scheduler; the space requirement at each node is $O(\log n)$ bits. The algorithm assumes a distinguished root node and nodes with unique identifiers.

In algorithm `MWCDS-S`, each node $i, 1 \leq i \leq n$, maintains the following (stored) variables:

- A boolean membership flag $\boldsymbol{s_i}$ indicating membership status of node i in a system state; at any time (system state) $\mathcal{S}$ is the current set of nodes with $s_i = 1$.

- A non-negative integer variable $\boldsymbol{d_i}$ that keeps track of the current distance of node i to *root* in any system state.
- A boolean flag $\boldsymbol{m_i}$; node i sets this flag to attempt to change membership status in $\mathcal{S}$.

Definition 2.2.1 *In any system state, for a node $i \neq root$, two logical variables are defined:*

$$\rho_i = \min\{d_j | j \in N(i)\} + 1$$

$$minM_i = \min\{j | j \in N[i] \wedge d_j = d_i \wedge m_j = 1\} \ where \min\{\} = null$$

Remark 2.2.1 *ρ_i estimates the distance of node i to root from the current content of the d-variables of its neighbors. $minM_i$ denotes the smallest ID node, say j, in the closed neighborhood of node i with equal d-values and true m flags; this is used to resolve the tie among competing nodes attempting to change membership status. Both logical variables are locally computable at node i in any system state.*

Definition 2.2.2 *In any system state, two additional Boolean predicates are defined for each node $i \neq root$:*

$$\texttt{enter}_i \stackrel{\text{def}}{\equiv} (s_i = 0) \wedge (\forall j \in N_{\leq}(i) : s_j = 0)$$

$$\texttt{leave}_i \stackrel{\text{def}}{\equiv} (s_i = 1) \wedge (\exists j \in N_{\leq}(i) : s_j = 1)$$

where $N_{\leq}(i)$ denotes the set of nodes $j \in N(i)$ with $d_j \leq d_i$.

The algorithm `MWCDS-S` constructs a minimal weakly connected dominating set (MWCDS) for graph G in an incremental fashion, i.e, when an MWCDS is computed for the subgraph induced by the nodes with distance of $\leq k$ to root ($1 \leq k < \Delta$), an MWCDS for the subgraph induced by the nodes with distance of $\leq k+1$ to root will be obtained in a finite number of steps. As the starting point, the root is initially set to be a member of MWCDS. In the incremental step, each non-root node looks at its neighbors with equal or smaller distance to root: if such a neighbor is already in the MWCDS, then it leaves set and otherwise it enters into the set if necessary. To avoid infinite loop in the incremental step, nodes with the same distance to root are enforced to change their memberships in sequential (implemented by stored m-variables). The complete pseudocode of algorithm `MWCDS-S` on node i, $1 \leq i \leq n$, is shown in Figure 2.1.

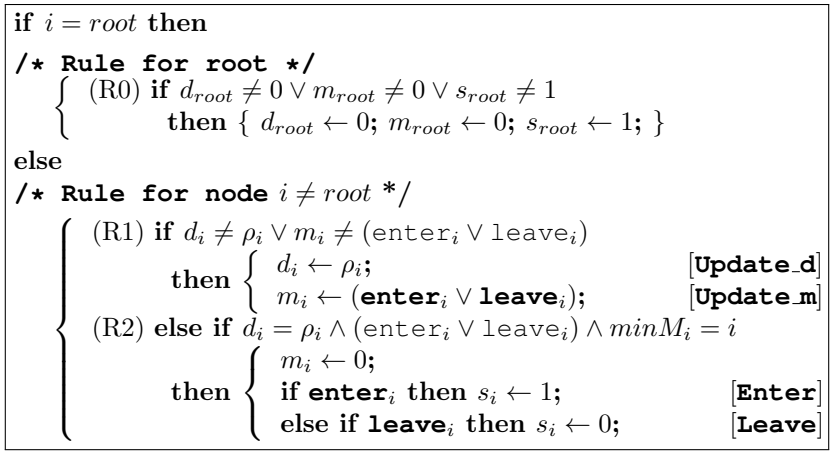

Figure 2.1: Algorithm MWCDS-S on node i, $1 \leq i \leq n$

Definition 2.2.3 *In any system state, a node is **privileged** if it is enabled by the algorithm to take action (make a move); the algorithm MWCDS-S **terminates** when no node is privileged.*

Definition 2.2.4 *A* membership move *of node $i \in V$ is defined to be a move that changes the membership of i in $\mathcal{S}$ (**Note:** both Enter and Leave moves are membership moves, while the Update moves, Update_d and Update_m, are not).*

Remark 2.2.2 *In a system state two adjacent nodes with equal d-values cannot make membership moves simultaneously in a synchronous step (a node i must have $minN_i = i$ to make a membership move and nodes have unique IDs).*

Correctness & Convergence

We first prove that $\mathcal{S}$ is a minimal weakly connected dominating set if algorithm MWCDS-S terminates; next, we show the algorithm, starting from an arbitrary state, always terminates in $O(n)$ steps using a synchronous scheduler.

Lemma 2.2.1 *If algorithm MWCDS-S terminates, then $d_{root} = 0$, $s_{root} = 1$, and for each node $i \neq root$, $d_i = \rho_i$.*

Proof: This lemma immediately follows from the fact that the root node is not privileged by rule R0 and other nodes are not privileged by rule R1 at termination (Definition 2.2.3). □

Lemma 2.2.2 *If algorithm MWCDS-S terminates, for each node $i \in V$, $m_i = 0$.*

Proof: Assume, by contradiction, there exist some node(s) i with $m_i = 1$. Consider the node j with minimum ID from among those nodes. (a) if $j = root$, then node j is privileged by rule R0, a contradiction; (b) if $j \neq root$, either $\texttt{enter}_j = 1$ or $\texttt{leave}_j = 1$ (otherwise node j is privileged by rule R1); thus node j is privileged by rule R2 by Lemma 2.2.1, a contradiction. □

Lemma 2.2.3 *If algorithm MWCDS-S terminates, then for each node $i \neq root$,* $\texttt{enter}_i = \texttt{leave}_i = 0$.

Proof: This immediately follows from Lemmas 2.2.1 and 2.2.2 and the fact that no node is privileged by rule R1 when the algorithm terminates. □

Lemma 2.2.4 *If algorithm MWCDS-S terminates, for each node $i \neq root$:*

(a) node i has at least one neighbor j such that $d_j = d_i - 1$.

(b) if $s_i = 0$, node i has at least one $\mathcal{S}$-neighbor(s) j with $d_j \leq d_i$;

(c) if $s_i = 1$, node i has at least one non-$\mathcal{S}$-neighbor(s) j with $d_j = d_i - 1$.

Proof: (a) Assume otherwise; then, $\rho_i \neq d_i$ (Definition 2.2.1) and node i is privileged by rule R1, a contradiction. (b) Since $\texttt{enter}_i = 0$ by Lemma 2.2.3, node i has at least one $\mathcal{S}$-neighbor j with $d_j \leq d_i$ by Definition 2.2.2; (c) Since $s_i = 1$ and $\texttt{leave}_i = 0$ by Lemma 2.2.3, node i does not have any neighbor j such that $s_j = 1$ and $d_j \leq d_i$ (Definition 2.2.2). Thus, the neighbor j of node i with $d_j = d_i - 1$ must have $s_j = 0$ or $j \notin \mathcal{S}$. □

Lemma 2.2.5 *If algorithm MWCDS-S terminates, then $\mathcal{S}$ is a weakly connected dominating set.*

Proof: $\mathcal{S}$ is dominating set (Lemma 2.2.4(b)) and root node is in $\mathcal{S}$ (Lemma 2.2.1). Consider an arbitrary node $i \in G$; we can construct a path to the node *root* by applying Lemma 2.2.4 repeatedly (G is connected); in any such path, no two consecutive nodes are out of $\mathcal{S}$ (Lemma 2.2.4); this implies that $\mathcal{S}$ is weakly connected. □

Theorem 2.2.1 *If algorithm MWCDS-S terminates, then $\mathcal{S}$ is a minimal weakly connected dominating set.*

Proof: $\mathcal{S}$ is a weakly connected dominating set (Lemma 2.2.5). To show that $\mathcal{S}$ is minimal, assume otherwise; there exists a node $i \in \mathcal{S}$ such that $\mathcal{S} - \{i\}$ is also a weakly connected dominating set.

Node i must then have a neighbor $j \in \mathcal{S} - \{i\}$ with $s_j = 1$ for $\mathcal{S} - \{i\}$ to be a dominating set. There are two possibilities: (a) $i \neq root$: then $\texttt{leave}_i$ and/or $\texttt{leave}_j$ is true by Definition 2.2.2, a contradiction (Lemma 2.2.3); (b) $i = root$: then i is privileged by rule R0, a contradiction (algorithm `MWCDS-S` has terminated). □

Definition 2.2.5 *In any system state a node is called* ***dead*** *if the node is never privileged again until termination.*

Lemma 2.2.6 *Starting in an arbitrary state, the root node will be dead after synchronous step 1 of execution.*

Proof: The lemma immediately follows from the fact that the root node corrects its d_{root}, m_{root} and s_{root} by executing rule R0 in step 1 if necessary. □

Lemma 2.2.7 *After k^{th} step of execution, $1 \leq k \leq \Delta$, (a) a node i at a distance $k-1$ from the root node (i.e., $\mu_i = dist(i, root) = k-1$) has $d_i = k-1$ and node i will never update its d_i again until termination* (d_i has stabilized at its correct value)*; (b) a node j at a distance $\geq k$ from the root node (i.e., $\mu_i \geq k$) has $d_j \geq k$* (d_j has not possibly stabilized to its correct value).

Proof: We prove by induction.

Basis Step: (a) is true for $k = 1$ by Lemma 2.2.6 and rule R0, and (b) is true because each node $j \neq root$ makes Update_d move in step 1 if necessary such that $d_j = \rho_j \geq 1$.

Induction Step: Assume, when $k = t$, (a) and (b) both are true, i.e., each node i with $\mu_i = t$ has $\rho_i = t$ and each node j with $\mu_j = t+1$ has $\rho_j \geq t+1$ at the end of t^{th} step (Definition 2.2.1). In step $t+1$, each node i with $\mu_i \geq t$ makes Update_d move, if necessary, such that $d_i = \rho_i$. Coupled with the fact that nodes at a distance $t-1$ to root do not change their d-variables after step t, we get that any node i with $\mu_i = t$ will never change its d_i after step $t+1$. Thus, both (a) and (b) are true for $k = t+1$. The lemma follows. □

Definition 2.2.6 *A system state is called a* ***d-legitimate*** *state where for each $i \in V$, $d_i = \mu_i$. In a d-legitimate state, we use n_k to denote the number of nodes i with $d_i = k$, where $0 \leq k \leq \Delta$, (**note:** $\sum_{k=0}^{\Delta} n_k = n$).*

Theorem 2.2.2 *Starting in an arbitrary state, the system enters in a d-legitimate state after $\Delta + 1$ steps, and remains in a d-legitimate state until termination* (no node will execute Update_d move again).

Proof: The proof immediately follows from Lemmas 2.2.6 and 2.2.7. □

Lemma 2.2.8 *In a d-legitimate state, if all nodes j with $d_j = \mu_j = k-1$, $1 \leq k \leq \Delta$, are dead, (a) any node i with $d_i = \mu_i = k$ can make at most two membership moves; (b) all nodes i with $d_i = \mu_i = k$ become dead in at most $4n_k + 1$ steps.*

Proof: (a) When i enters $\mathcal{S}$, $s_i = 1$ and $\texttt{leave}_i = 0$ (Definition 2.2.2 and Remark 2.2.2); $\texttt{enter}_j = 0$ for each neighbor j of i with $d_j = d_i$ and hence cannot make any more Enter move; thus $\texttt{leave}_i$ remains 0, i.e., node i cannot make a Leave move any more. Thus, if a node's first membership move is Enter, then it will not make a membership move again. If its first membership move is Leave, then any next membership move must be Enter; it cannot make another membership move.

(b) All nodes i with $d_i = k$ make at most $2n_k$ membership moves in total by part (a) and Definition 2.2.6. In the worst case, each membership move takes one step (Remark 2.2.2); In between two such membership moves, each of such nodes will execute Update_m moves in one step, if necessary. The proof follows. □

Theorem 2.2.3 *Starting in an arbitrary state, algorithm MWCDS-S terminates in a minimal weakly connected dominating set in $O(n)$ steps.*

Proof: Starting in an arbitrary state the system reaches a d-legitimate state in at most $\Delta + 1$ steps (Theorem 2.2.2). In a d-legitimate state, all nodes in V become dead in at most $\sum_{k=0}^{\Delta}(4n_k + 1) = 4n + \Delta$ steps (Lemmas 2.2.6 and 2.2.8). Thus, starting in an arbitrary state the algorithm terminates in at most $4n + \Delta + \Delta + 1 = O(n)$ steps. $\mathcal{S}$, at that time is a minimal weakly connected set of V (Theorem 2.2.1). □

An Illustrative Example

Figure 2.2 shows an execution sequence of the algorithm MWCDS-S on a graph with $n = m = 7$; a shaded circle denotes the node in $\mathcal{S}$; node 2 is assumed to be the root node. The execution of algorithm MWCDS-S is managed by a synchronous scheduler, that selects all privileged nodes in a system state to move synchronously and atomically in each step.

Figure 2.2(a) is an arbitrary initial state, where nodes 1, 3, 4, 5, 6, 7 are privileged by the rule R1 and node 2 is privileged by the rule R0. Successive figures show the transition of the global states as the synchronous scheduler selects all privileged nodes to move in each step. After step 5,

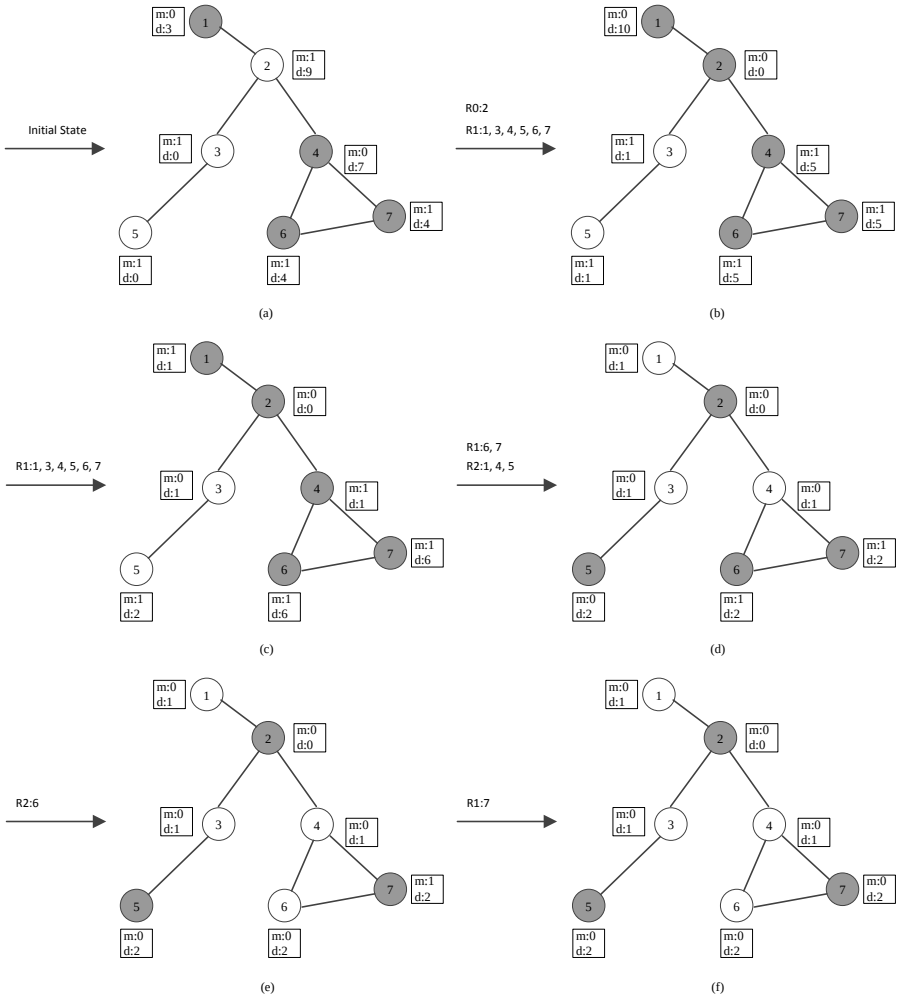

Figure 2.2: An example to illustrate the execution of algorithm MWCDS−S

no node is privileged (the algorithm terminates); we get the minimal weakly connected dominating set $\{2, 5, 7\}$.

2.3 Algorithm MWCDS−C

Algorithm MWCDS−C is a semi-uniform (all nodes are identical and anonymous except one distinguished root node called r) self-stabilizing algorithm that, starting from an arbitrary illegitimate state, computes a minimal weakly connected dominating set using an unfair central scheduler. Note that node numbers are used for reference purposes only.

In algorithm MWCDS−C, each node $i, 1 \leq i \leq n$, maintains the following (stored) variables:

- A boolean membership flag $\boldsymbol{s_i}$ that indicates membership status of node i in $\mathcal{S}$ in a system state; in any system state $\mathcal{S}$ is $\{i \in V : s_i = 1\}$.
- A non-negative integer variable $\boldsymbol{d_i}$ that keeps track of the current distance of node i to r in any system state.
- A non-negative integer clock variable $\boldsymbol{\mathcal{C}_i}$ which is incremented in modulo $n+1$, where $n = |V|$.

Definition 2.3.1 *In any system state, for a node $i \neq r$, the logical variable $\boldsymbol{\rho_i}$ estimates the distance of node i to the distinguished node r from the current contents of the d-variables of its neighbors:*

$$\rho_i = \min\{d_j | j \in N(i)\} + 1$$

Definition 2.3.2 *In any system state, two Boolean predicates are defined for each node $i \neq r$:*

$$\texttt{enter}_i \stackrel{\text{def}}{\equiv} (s_i = 0) \wedge (\forall j \in N_{\leq}(i) : s_j = 0)$$

$$\texttt{leave}_i \stackrel{\text{def}}{\equiv} (s_i = 1) \wedge (\exists j \in N_{\leq}(i) : s_j = 1)$$

where $N_{\leq}(i)$ denotes the set of nodes $j \in N(i)$ with $d_j \leq d_i$.

Note: ρ_i, $\texttt{enter}_i$ and $\texttt{leave}_i$ are locally computable at node i in any system state.

The underlying approach is simple. The node r enters $\mathcal{S}$ if it is not in $\mathcal{S}$; any other node $i \neq r$ looks at its neighbors with equal or smaller distance to node r: if such a neighbor is already in $\mathcal{S}$, then it leaves $\mathcal{S}$ and otherwise it enters $\mathcal{S}$ if necessary. A counter technique (implemented by $\mathcal{C}$-variables) guarantees that each node $i \in V$ is privileged to update its s_i and d_i within an interval of at most n^3 steps (the proof is given later). The complete pseudocode of algorithm MWCDS−C is shown in Figure 2.3.

Definition 2.3.3 *(a) A node is **privileged** iff it is enabled to move by the algorithm; (b) A global system state is **legitimate** iff any node $i \in V$ can make only Progress moves, i.e., (1) $d_r = 0$ and $s_r = 1$, and (2) for a node $i \neq r$: $d_i = \rho_i$ and $\texttt{enter}_i = \texttt{leave}_i = 0$.*

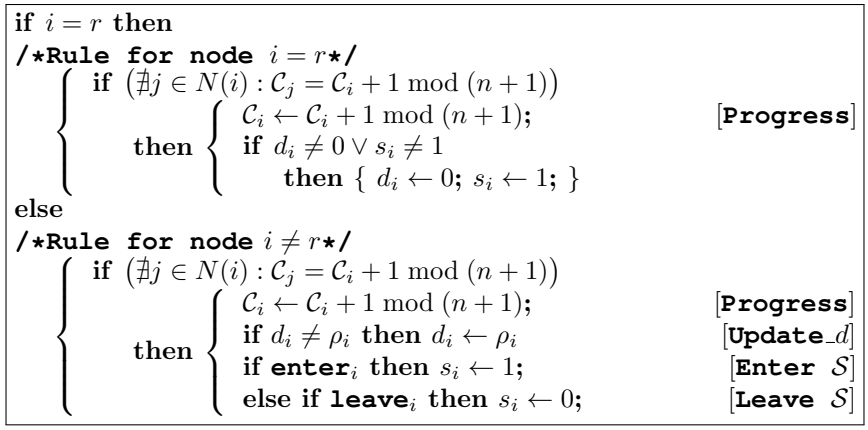

Figure 2.3: Algorithm MWCDS-C on node i, $1 \leq i \leq n$

Definition 2.3.4 *A move by a privileged node i is called a **membership move** if the move changes the membership of i in $\mathcal{S}$ (**Note:** both Enter and Leave moves are membership moves, while Progress and Update_d moves are not).*

Lemma 2.3.1 *In any system state, there exists at least one privileged node (i.e., the algorithm never terminates).*

Proof: Assume, by contradiction, algorithm MWCDS-C terminates (no node is privileged); then for each node i, there exists $j \in N(i)$ such that $\mathcal{C}_j = \mathcal{C}_i + 1 \bmod (n+1)$. This implies $n+1$ different values of the clock variables, and hence $n+1$ different nodes, a contradiction. □

Theorem 2.3.1 *Starting in an illegitimate state, if the system enters in a legitimate state, the system makes transitions between legitimate states thereafter.*

Proof: A privileged node in a legitimate state can make only a Progress move; it cannot make any Enter $\mathcal{S}$, Leave $\mathcal{S}$ or Update_d move (Definition 2.3.3). Thus, if the system enters in a legitimate state, only $\mathcal{C}$-variables will change by the Progress moves and the algorithm makes transitions between legitimate states thereafter. □

Correctness & Convergence

We first prove that $\mathcal{S}$ is a minimal weakly connected dominating set if algorithm MWCDS-C stabilizes in a legitimate state (Definition 2.3.3); then we show the algorithm stabilizes in $O(n^4)$ steps using an unfair central scheduler.

Lemma 2.3.2 *If algorithm* `MWCDS-C` *stabilizes in a legitimate state, for each node* $i \neq r$*:*

(a) node i *has at least one neighbor* j *such that* $d_j = d_i - 1$*;*

(b) if $s_i = 0$*, then node* i *has* $\mathcal{S}$*-neighbor(s)* j *with* $d_j \leq d_i$*;*

(c) if $s_i = 1$*, then node* i *has non-*$\mathcal{S}$*-neighbor(s)* j *with* $d_j = d_i - 1$*.*

Proof: (a) Assume otherwise; then $d_i \neq \rho_i$ (Definition 2.3.1), a contradiction with Definition 2.3.3. (b) $\texttt{enter}_i = 0$ by Definition 2.3.3, thus node i has at least one $\mathcal{S}$-neighbor(s) j with $d_j \leq d_i$ by Definition 2.3.2; (b) Since $s_i = 1$ and $\texttt{leave}_i = 0$ by Definition 2.3.3, node i does not have any neighbor j such that $s_j = 1$ and $d_j \leq d_i$ (Definition 2.3.2). Thus, the neighbor j of node i with $d_j = d_i - 1$ must have $s_j = 0$ or $j \notin \mathcal{S}$. □

Lemma 2.3.3 *If algorithm* `MWCDS-C` *stabilizes in a legitimate state, then* $\mathcal{S}$ *is a weakly connected dominating set.*

Proof: $\mathcal{S}$ is dominating set by Lemma 2.3.2(b) and the r node is in $\mathcal{S}$ (Definition 2.3.3). Consider an arbitrary node i in G, we can construct a path to the node r by applying Lemma 2.3.2 repeatedly (G is connected); in any such path, no two consecutive nodes are out of $\mathcal{S}$ (Lemma 2.3.2); this implies that $\mathcal{S}$ is weakly connected. □

Theorem 2.3.2 *If algorithm* `MWCDS-C` *stabilizes in a legitimate state, then* $\mathcal{S}$ *is a minimal weakly connected dominating set.*

Proof: $\mathcal{S}$ is a weakly connected dominating set (Lemma 2.3.3). To show that $\mathcal{S}$ is minimal, assume otherwise; there exists some node(s) $i \in \mathcal{S}$ such that $\mathcal{S} - \{i\}$ is also a weakly connected dominating set, which means i is adjacent to some node(s) j with $s_j = 1$. There are two possibilities: (a) if $i \neq r$, then $\texttt{leave}_i$ and/or $\texttt{leave}_j$ is true by Definition 2.3.2; a contradiction with Definition 2.3.3; (b) if $i = r$, the removal of node i would make $s_r = 0$, a contradiction with Definition 2.3.3. □

Lemma 2.3.4 *If a node* $i \in V$ *does not move, then any node* j *with* $dist(i, j) = \beta$ *can move at most* $\beta \times n$ *times where* $1 \leq \beta \leq \Delta$*.*

Proof: If node i does not move, then $\mathcal{C}_i$ stays unchanged. We prove the lemma by induction:
Basis Step: If $\beta = 1$, $\mathcal{C}_j$ will reach $\mathcal{C}_i - 1 \bmod (n + 1)$ in at most n moves and cannot be privileged again until node i moves.

Induction Step: Assume the claim is true for $\beta = t$, i.e., any node j with $dist(i,j) = t$ can move at most $t \times n$ times if node i does not move. Consider a node k, $dist(i,k) = t+1$; k must have a neighbor j such that $dist(i,j) = t$. If node j does not move, node k can move at most n times; node k can move only when $\mathcal{C}_j \neq \mathcal{C}_k + 1 \mod (n+1)$ and both nodes j and k advance their clock variables by 1 in a move; thus, node k can make at most $t \times n + n = (t+1) \times n$ moves if node i does not move. □

Lemma 2.3.5 *Each node $i \in V$ moves at least once in an interval of n^3 steps of execution.*

Proof: Assume, by contradiction, node i does not move. Then, nodes in $V - \{i\}$ can together make at most $\Delta \times n^2$ moves (Lemma 2.3.4), $1 \leq \Delta \leq n-1$ and become unprivileged. Node i becomes the only privileged node (Lemma 2.3.1) and has to be picked up by the central scheduler to move. □

Definition 2.3.5 *In any system state, a node $i \in V$ is called (a) **d-stable** if (i) $i = r$ and $d_r = 0$, or (ii) $i \neq r$, i is never enabled to make an Update_d move, and it has a d-stable neighbor j such that $d_j = d_i - 1$; (b) **consistent** if i is d-stable and is never enabled to make any membership move.*

Remark 2.3.1

1. *A d-stable node i has $\mu_i = d_i$.*
2. *The distinguished node r is consistent (and d-stable) if $d_r = 0$ and $s_r = 1$.*
3. *A consistent node, when privileged, can only be enabled to make Progress moves.*

Lemma 2.3.6 *Starting in any illegitimate state, the distinguished node r will be consistent in at most n^3 steps of execution.*

Proof: This lemma immediately follows from the fact that node r moves in at most n^3 steps by Lemma 2.3.5 and sets $d_r = 0$ and $s_r = 1$; thus, it can make only Progress moves thereafter. □

Lemma 2.3.7 *Starting in an arbitrary state, for $1 \leq k \leq \Delta$, a node i with $\mu_i = k-1$ will be d-stable and a node j with $\mu_j \geq k$ will have $d_j \geq k$, in at most kn^3 steps.*

Proof: We prove by induction.

Basis Step: For $k = 1$, node r is the only node with $\mu_i = 0$ and in at most n^3 steps, node r will be d-stable (Definition 2.3.5, Remark 2.3.1 and Lemma 2.3.6); any node $i \neq r$ will move at least once and adjust $d_i \geq 1$ (Lemma 2.3.5 and all d variables are non-negative).

Induction Step: Assume the claim is true for $k = t$, i.e., a node i with $\mu_i = t - 1$ is d-stable and a node j with $\mu_j \geq t$ has $d_j \geq t$ in at most tn^3 steps. In at most n^3 steps thereafter, each node will be privileged at least once: each node with $\mu_i = t$ (that has at least one d-stable neighbor j with $\mu_j = d_j = t - 1$) will execute an Update_d move to have $d_i = \rho_i = d_j + 1 = t$ and become d-stable ; each node with $\mu_i \geq t + 1$ will execute an Update_d move to have $d_i = \rho_i \geq t + 1$ (each such i has all its neighbors j with $d_j \geq t$ by assumption). Thus the claim is true for $k = t + 1$. The lemma follows. □

Definition 2.3.6 *A system state is called* ***d-legitimate*** *state where each $i \in V$ is d-stable (i.e., $d_i = \mu_i$). In a d-legitimate state, we use n_k to denote the number of nodes i with $d_i = k$, where $0 \leq k \leq \Delta$ (**note**: $\sum_{k=0}^{\Delta} n_k = n$).*

Theorem 2.3.3 *Starting in an arbitrary state, the system enters in a d-legitimate state after $(\Delta + 1)n^3$ steps, and remains in a d-legitimate state until the algorithm stabilizes in a legitimate state* (no node will execute Update_d move again).

Proof: This theorem immediately follows from Lemmas 2.3.6 and 2.3.7. □

Remark 2.3.2 *Theorem 2.3.3 and all previous definitions, lemmas and theorems are valid for an unfair distributed scheduler (hence for central scheduler as well) since the arguments used in arriving at the results do not depend on the number of active privileged nodes in any particular step of execution.*

Lemma 2.3.8 *In a d-legitimate state using an unfair central scheduler, if all nodes j with $\mu_j = k-1$, $1 \leq k \leq \Delta$, are consistent, (a) any node i with $d_i = \mu_i = k$ can make at most two membership moves; (b) all nodes i with $d_i = \mu_i = k$ become consistent in at most $(2n_k)n^3$ steps.*

Proof: (a) Assume a node i is privileged to execute Enter $\mathcal{S}$ move; $\texttt{enter}_i = 1$, i.e., $s_i = 0 \wedge (\forall j \in N_{\leq}(i) : s_j = 0)$ before the move; after the move, $s_i = 1$ and $\texttt{leave}_i = 0$ (Definition 2.3.2) since no neighbor j of i moves in the step (central scheduler); $s_j = 0$ and $\texttt{enter}_j = 0$ for each neighbor j of i with $d_j \leq d_i$ and hence cannot make any more Enter moves; thus $\texttt{leave}_i$ remains 0, i.e., node i cannot make a Leave move. Thus, if a node's first membership move is Enter, then it will not make a membership move again; if its first membership move is Leave, then any next membership move must be Enter, after which, it cannot make another membership move.

(b) All nodes i with $d_i = \mu_i = k$ become consistent in at most $2n_k$ moves by Part(a) and Definition 2.3.6. Each move needs at most n^3 steps in the worst case (Lemma 2.3.5). The lemma holds. □

Theorem 2.3.4 *Starting in an arbitrary state, algorithm* `MWCDS-C` *stabilizes in a minimal weakly connected dominating set in $O(n^4)$ steps using an unfair central scheduler.*

Proof: Starting in an arbitrary state, the system reaches a d-legitimate state in at most $(\Delta + 1)n^3$ steps (Theorem 2.3.3). In a d-legitimate state, all nodes in V become consistent in at most $\sum_{k=0}^{\Delta}(2n_k)n^3 = 2n^4$ steps. $\mathcal{S}$ at that time is a minimal weakly connected dominating set (Definition 2.3.3 and Theorem 2.3.2). □

2.4 Algorithm `MWCDS-D`

Algorithm `MWCDS-C`, developed in the previous section, is a semi-uniform self-stabilizing algorithm (only one node is designated as a distinguished root node and all other nodes are anonymous) that works with an unfair central scheduler with time complexity of $O(n^4)$ steps and space requirement of $O(\log n)$ bits at each node. In this section, we further assume that the nodes in the graph have unique IDs and proposes a modification to the previous algorithm `MWCDS-C` such that the resulting algorithm `MWCDS-D` works with an unfair distributed scheduler with the same asymptotic time and space complexities.

Note that authors in [27] provide a mechanism that can be used to convert any anonymous self-stabilizing algorithm (distance-1 model) working with an unfair central scheduler to one that works with an unfair distributed scheduler with an $O(n)$ slowdown, where n is the number of nodes in the graph. The basic idea of [27] is to use node IDs to avoid simultaneous rule executions at adjacent nodes and thus, the unfair distributed scheduler becomes equivalent to an unfair central scheduler with an $O(n)$ slowdown. A close observation of the algorithm `MWCDS-C` reveals that a slight modification can be done using the distinct node IDs and an extra one bit binary flag at each node such that the resulting algorithm (called algorithm `MWCDS-D`) stabilizes under an unfair distributed scheduler with a constant slowdown. The underlying approach is simple: (a) note that starting in an arbitrary state, the algorithm `MWCDS-C` reaches a d-legitimate state in $O(n^4)$ steps using an unfair distributed scheduler (Remark 2.3.2); (b) Next, once a d-legitimate state is reached,

to reach a legitimate state using an unfair distributed scheduler, distinct node IDs are used to achieve local mutual exclusion such that the simultaneous membership moves at the neighboring nodes with the same d-values are avoided.

Algorithm MWCDS-D requires that each node i, $1 \leq i \leq n$, maintain an extra boolean flag $\boldsymbol{m_i}$; node i sets this flag to indicate intent to change membership status in $\mathcal{S}$.

Definition 2.4.1 *In any system state, a logical variable* $\boldsymbol{minM_i}$*, for a node* $i \neq r$*, denotes the smallest ID node in the closed neighborhood of node* i *with* $d = d_i$ *and* $m = 1$*, i.e.,*

$$minM_i = \min\{j | j \in N[i] \wedge d_j = d_i \wedge m_j = 1\} \text{ where } \min\{\} = null$$

Algorithm MWCDS-D includes one extra Update_m move that sets the intent of node i to change its membership status (in $\mathcal{S}$) before it can actually changes the membership in the next move; if more than one adjacent nodes have their intent flag set in any step, the tie is resolved by adjusting the condition for the membership move in the next step by using $minM_i$. The complete pseudocode for algorithm MWCDS-D at node i, $1 \leq i \leq n$, is shown in Figure 2.4 for convenience.

```
if i = r then
/*Rule for node i = r*/
    if ∄j ∈ N(i) : C_j = C_i + 1 mod (n+1)
        then { C_i ← C_i + 1 mod (n+1);                          [Progress]
               if d_i ≠ 0 ∨ m_i ≠ 0 ∨ s_i ≠ 1
                   then { d_i ← 0; m_i ← 0; s_i ← 1; } }
else
/*Rule for node i ≠ r*/
    if ∄j ∈ N(i) : C_j = C_i + 1 mod (n+1)
        then { C_i ← C_i + 1 mod (n+1);                          [Progress]
               if d_i ≠ ρ_i ∨ m_i ≠ (enter_i ∨ leave_i)
                   then { d_i ← ρ_i;                             [Update_d]
                          m_i ← (enter_i ∨ leave_i); }           [Update_m]
               else if d_i = ρ_i ∧ (enter_i ∨ leave_i) ∧ minM_i = i
                   then { m_i ← 0;
                          if enter_i then s_i ← 1;               [Enter]
                          else if leave_i then s_i ← 0; }        [Leave]
```

Figure 2.4: Algorithm MWCDS-D on node i, $1 \leq i \leq n$

Remark 2.4.1 *In a system state two adjacent nodes (not node* r*) with equal* d*-values cannot make membership moves simultaneously in a step (a node* $i \neq r$ *must have* $minM_i = i$ *to make a mem-*

bership move and nodes have unique IDs).

Lemma 2.4.1 *In a d-legitimate state, under an unfair distributed scheduler, if all nodes j with $\mu_j = k-1$, $1 \leq k \leq \Delta$, are consistent, (a) any node i with $d_i = \mu_i = k$ can make at most two membership moves; (b) all nodes i with $d_i = \mu_i = k$ become consistent in at most $(4n_k + 1)n^3$ steps.*

Proof: (a) Assume a node i is privileged to execute Enter $\mathcal{S}$ move; $\texttt{enter}_i = 1$, (i.e., $s_i = 0 \wedge (\forall j \in N_{\leq}(i) : s_j = 0)$), $m_i = 1$, and $minM_i = i$ before the move. After the move, $s_i = 1$ and $\texttt{leave}_i = 0$ since no $j \in N_{\leq}(i)$ can make any membership move in the step (Remark 2.4.1). Thus, after the move, $\forall j \in N_{\leq}(i) : s_j = 0$ and $\texttt{enter}_j = 0$ and will remain so in all subsequent moves and hence $\texttt{leave}_i$ remains 0; i.e., node i cannot make a Leave move ever again. Thus, if a node's first membership move is Enter, then it will not make a membership move again; if its first membership move is Leave, then any next membership move must be Enter, after which, it cannot make another membership move.

(b) All nodes i with $d_i = \mu_i = k$ collectively make at most $2n_k$ membership moves by Part (a) and Definition 2.3.6. Each membership move needs at most n^3 steps in the worst case (Theorem 2.3.3, Lemma 2.3.5 and Remark 2.3.2); in between two such membership moves, each node may execute one Update_m move in the worst case taking at most n^3 steps if necessary (Theorem 2.3.3 and Lemma 2.3.5). The lemma holds. □

Theorem 2.4.1 *Starting in an arbitrary state, algorithm* `MWCDS-D` *stabilizes in a minimal weakly connected dominating set in $O(n^4)$ steps using an unfair distributed scheduler.*

Proof: Starting in an arbitrary state, the system reaches a d-legitimate state in at most $(\Delta+1)n^3 = O(n^4)$ steps (Theorem 2.3.3 and Remark 2.3.2). In a d-legitimate state, each node in V becomes consistent in at most $\sum_{k=0}^{\Delta}(4n_k + 1)n^3 = (4n + \Delta)n^3 = O(n^4)$ steps (Lemma 2.4.1). $\mathcal{S}$ at that time is a minimal weakly connected dominating set (Definition 2.3.3, Theorem 2.3.2 and Remark 2.3.2). □

Chapter 3

Selection of Influential Users in Social Networks Using Self-Stabilizing Algorithm

3.1 Influential Users Selection in Social Network

The study of influence propagation in social networks has recently received a great deal of attention [45, 5, 2, 4]. The influence propagation is based on the observation that a user's specific behaviors and/or opinions in general affect those of his/her social contacts (or friends). It is regarded as an important tool to spread specific information in social networks and thereby to influence public behaviors and opinions, such as curbing social problems and viral marketing. Not all users affect their contacts or friends equally; selection of *influential users* is a critical component in social network influence propagation. Consider a social network with drinking problem, where binge drinkers and abstainers have positive and negative influence on their friends respectively. An abstainer may become a binge drinker if most of his/her friends are binge drinkers and vice versa. To ameliorate the drinking problem in such a network, it is advisable to select a subset of individuals (influential users) to participate in the intervention program due to the limitations of education recourses or budget; the selection of influential users is expected to ensure that the effect of the program would be spread over the entire network while minimizing the cost. Most existing studies focus on how to

maximize the influence of an initially selected set of influential users, paying less attention on how the selection of influential users could maximize the speed of the propagation [37, 38, 55]. Thus, it is important to investigate how one can select an optimum set of influential users to maximize the speed of influence propagation in social networks.

A social network can be modeled by a symmetric graph $G = (V, E)$. A set $\mathcal{S} \subseteq V$ is a positive influence dominating set (PIDS) iff each node $i \in V$ is dominated by at least $\lceil \delta_i/2 \rceil$ nodes, where δ_i is the degree of node i. The authors in [49] formulated the concept of PIDS, and proposed to select the nodes in PIDS as *influential nodes*. PIDS can be easily applied to alleviate the drinking problem; if the nodes in PIDS participate in the intervention program, each node i is adjacent to at least $\lceil \delta_i/2 \rceil$ influential nodes (i.e., each binge drinker have more abstainer friends than binge drinking friends), and hence the influence propagation time is 1-hop (or distance 1). Authors in [50] presented a greedy algorithm to compute PIDS with execution time complexity of $O(n^3)$, where n is the number of nodes in the network graph. Subsequently, authors in [40] proposed a new greedy algorithm with time complexity of $O(n^2)$; the experimental results show that the size of PIDS generated in [40] is smaller than the one in [50]. Recently, [17] proposed an $O(n)$ time algorithm to compute PIDS in tree network graphs.

All of these PIDS algorithms require the global information of the entire graph, i.e., the topology of the social network, which is unavailable for an ordinary user due to the privacy protection policy in social networking platforms. Although the social network platform, such as Facebook, is able to access the topology of the entire network graph, the servers running PIDS algorithms may become bottlenecks due to their limited computation and network capacities when the network expands over time.

While self-stabilization has been widely used in designing fault tolerant distributed algorithms in communication networks [28], its application to solve social network problems is very recent; a self-stabilizing algorithm to compute minimal PIDS has been proposed in [51]; the algorithm requires that participating nodes need only the distance-1 information, i.e., the information on its immediate neighbors (*distance-1 model*); note that the distance-1 information in any social network platform can be easily obtained from user's contacts (i.e., friends). It assumes unique node IDs and a central scheduler. The space requirement at each node is $O(n \log n)$ bits. The run time complexity of this algorithm is indeed exponential in the number of nodes in the graph; an example execution of this algorithm [51] is provided in section 3.3 to show that it requires $O(2^{n+1})$ steps to

stabilize. PIDS approach is based on two assumptions: influence between two neighboring nodes is symmetric and each neighbor of a node i has uniform influence on i.

However, we observe that (1) the mutual influence between two neighboring nodes i and j in a social setting is in general asymmetric; and (2) each node has different tolerance level (sensitivity) towards neighbor's influences. In addition, since the algorithm works only for a central scheduler, it is not directly applicable on a social network platform where the operations are inherently asynchronous in nature.

In this chapter, a generalized approach to model influence in social networks is presented. The network is modeled as a weighted directed graph such that (1) each node $i \in V$ has a target integer t_i denoting the level of tolerance or sensitivity; and (2) each directed edge $i \rightarrow j$ is associated with a weight $w_{i,j}$ indicating the degree of influence of node i on its neighbor j. A set $\mathcal{S} \subseteq V$ is a *weighted positive influence dominating set (WPIDS)* iff each node $i \in V$ has $\sum_{j \in N(i) \cap \mathcal{S}} w_{j,i} \geq t_i$. A WPIDS $\mathcal{S}$ is called *minimal* iff there does not exist a node $i \in \mathcal{S}$ such that the set $\mathcal{S} - \{i\}$ is a WPIDS. It is to be noted that PIDS is a special case of WPIDS.

(1) The nodes in minimal WPIDS are proposed to be used as *influential nodes*, which overcomes the drawbacks of the PIDS approach, i.e., it accommodates the asymmetric influences between two neighboring users and different sensitivities of users to influences.

(2) The first $O(n^3)$ self-stabilizing algorithm is proposed to compute minimal WPIDS in an arbitrary social network graph; the algorithm works with an unfair distributed run time scheduler that is closest to the asynchronous nature of activities in a social network.

3.2 Minimal Weighted Positive Influence Dominating Set Algorithm

3.2.1 Model and Terminology

A social network can be modeled by a directed graph $G = (V, E)$, where V is the set of nodes (users), $|V| = n$, and E is the set of edges, $|E| = m$. If nodes i and j are friends, then both directed edges $i \rightarrow j$ and $j \rightarrow i$ belong to E. For a node i, $N(i)$, its *open neighborhood*, denotes the set of nodes adjacent to node i; $N[i] = N(i) \cup i$ denotes its *closed neighborhood*; δ_i denotes the degree

of node i. Assume that (1) each node $i \in V$ has a non-negative target integer t_i indicating the level of tolerance or sensitivity of node i; and (2) each directed edge $i \rightarrow j$ has a non-negative weight $w_{i,j}$ indicating the degree of influence from node i on its neighbor j. A set $\mathcal{S} \subseteq V$ is a *weighted positive influence dominating set* iff each node $i \in V$ has $\sum_{j \in N(i) \cap \mathcal{S}} w_{j,i} \geq t_i$. A weighted positive influence dominating set $\mathcal{S}$ is called *minimal* iff there does not exist a node $i \in \mathcal{S}$ such that the set $\mathcal{S} - \{i\}$ is a weighted positive influence dominating set. To ensure the existence of a minimal weighted positive influence dominating set, each node i is assumed to have $\sum_{j \in N(i)} w_{j,i} \geq t_i$.

Propagation Model: The influence propagation is based on the observation that a user's opinions or behaviors may affect the ones of his/her social contact (or friends), and further affect public opinions and behaviors. It is to be noted that the influence between two friends are asymmetric and different users have different sensitivities to the influence from their friends. If the amount of influence from node i's neighbors exceeds a threshold (i.e, t_i), then we assume node i is convinced about the same opinion or tends to show the same behavior as its neighbors.

Execution Model: The proposed algorithm does not need to know the size of the network graph. A node is privileged in a given system state iff it is enabled to move by any one of the rules of the algorithm; the algorithm terminates (stabilizes) in a system state when no node is privileged. The algorithm assumes state-reading model (as opposed to read/write atomicity model [18]), as is more customary with self-stabilizing algorithms for graph problems [28]. In order to analyze the correctness of the algorithm and its time complexity under a worst-case scenario, we assume (a) the execution of the algorithm at each node is managed by a central scheduler, that selects a privileged node in a system state to move in each step; and (b) each node knows the local states of nodes within its distance-2 (*distance-2 model*). Then, section 3.2.3 shows that the proposed algorithm can be converted to a distance-1 model, self-stabilizing algorithm that terminates with a minimal weighted positive influence dominating set in $O(n^3)$ steps using a distributed scheduler, that selects non-empty subset of the privileged nodes in a system state to move in each step.

3.2.2 Algorithm Using Distance-2 Model and Central Scheduler

To design a self-stabilizing algorithm for any graph problem, some predicate(s) needs to be defined at each node such that the node can then determine if it is locally legitimate (with respect to the given global system state of the problem at hand). For the ease of the algorithm development, distance-2 model is used to design the minimal weighted positive influence dominating set algorithm.

In the proposed self-stabilizing minimal weighted positive influence dominating set algorithm (called algorithm MWPIDS), each node $i, 1 \leq i \leq n$, maintains a boolean flag $\boldsymbol{s_i}$ denoting the membership of node i; node i is called $\boldsymbol{\mathcal{S}}$**-node** iff $s_i = 1$; in any system state $\mathcal{S}$ is the current set of $\mathcal{S}$-nodes.

Definition 3.2.1

1. *A node i is called* ***legal*** *in a (global) system state if $\sum_{j \in N(i) \cap \mathcal{S}} w_{j,i} \geq t_i$.*
2. *A system state is* ***legitimate*** *if $\mathcal{S}$ denotes a minimal weighted positive influence dominating set.*

The basic idea of algorithm MWPIDS is as follows: a node $i \notin \mathcal{S}$ enters $\mathcal{S}$ iff there exists at least one illegal node $j \in N(i)$, i.e., $\sum_{k \in N(j) \cap \mathcal{S}} w_{k,j} < t_j$ (Definition 3.2.1); a node $i \in \mathcal{S}$ leaves $\mathcal{S}$ iff each node $j \in N(i)$ has $\sum_{k \in N(j) \cap \mathcal{S}} w_{k,j} \geq t_j + w_{i,j}$.

Remark 3.2.1 *Consider any node i, $1 \leq i \leq n$, the legality of each $j \in N(i)$, i.e., $\sum_{k \in N(j) \cap \mathcal{S}} w_{k,j}$, is locally computable at node i in distance-2 model in any system state.*

The complete pseudo code of algorithm MWPIDS is shown in Figure 3.1.

RA : **if** $s_i = 0 \wedge \left(\exists j \in N(i) : \sum_{k \in N(j) \cap \mathcal{S}} w_{k,j} < t_j\right)$
  then $s_i \leftarrow 1$; **[Enter]**
RB : **if** $s_i = 1 \wedge \left(\forall j \in N(i) : \sum_{k \in N(j) \cap \mathcal{S}} w_{k,j} \geq t_j + w_{i,j}\right)$
  then $s_i \leftarrow 0$; **[Leave]**

Figure 3.1: Algorithm MWPIDS on node i, $1 \leq i \leq n$

Lemma 3.2.1 *If a node i is legal, i.e., $\sum_{j \in N(i) \cap \mathcal{S}} w_{j,i} \geq t_i$, then it remains legal under a central scheduler.*

Proof: If any node $k \in N(i)$ leaves $\mathcal{S}$, then $\sum_{j \in N(i) \cap \mathcal{S}} w_{j,i} \geq t_i$ by rule RB; the lemma holds. □

Lemma 3.2.2 *If a node leaves $\mathcal{S}$, it will never move again under a central scheduler.*

Proof: After node i leaves $\mathcal{S}$, all nodes $j \in N(i)$ are legal by rule RB; by Lemma 3.2.1 node i can not make Enter move again. □

Theorem 3.2.1 *The system can make at most $2n$ moves under a central scheduler.*

Proof: If a node's first move is Leave, then it will not make another move again by Lemma 3.2.2. If its first move is Enter, then any next move must be Leave; it cannot make another move. □

Theorem 3.2.2 *If algorithm* `MWPIDS` *terminates, then $\mathcal{S}$ is a minimal weighted positive influence dominating set.*

Proof: We first show that $\mathcal{S}$ is a weighted positive influence dominating set. Assume, by contradiction, $\mathcal{S}$ is not weighted positive influence dominating, i.e., there exists a node j such that $\sum_{k \in N(j) \cap \mathcal{S}} w_{k,j} < t_j$ (i.e., illegal); thus each node $i \in N(j) - \mathcal{S}$ is privileged to enter $\mathcal{S}$ by rule RA, a contradiction. Thus $\mathcal{S}$ is a weighted positive influence dominating set.

To show $\mathcal{S}$ is minimal we use contradiction again. Assume there exists a node $i \in \mathcal{S}$ such that $\mathcal{S} - \{i\}$ is a weighted positive influence dominating set. Since $i \in \mathcal{S}$ (i.e., $s_i = 1$), there must exist a node $j \in N(i)$ with $\sum_{k \in N(j) \cap \mathcal{S}} w_{k,j} < t_j + w_{i,j}$; the removal of i would make node j illegal, a contradiction. Thus $\mathcal{S}$ is minimal. □

Theorem 3.2.3 *Algorithm* `MWPIDS` *generates a minimal weighted positive influence dominating set and terminates in $O(n)$ steps using distance-2 model and a central scheduler.*

Proof: This follows from Theorems 3.2.1 and 3.2.2. □

3.2.3 Algorithm Using Distance-1 Model and Distributed Scheduler

The algorithm `MWPIDS`, developed in the previous section, is a self-stabilizing algorithm using distance-2 model under an unfair central scheduler; it has time complexity of $O(n)$ steps and space requirement of $O(\log n)$ bits at each node. In this section, the transformer in [47] is used to convert algorithm `MWPIDS` to a distance-1 model algorithm using a distributed scheduler, called algorithm `MWPIDS1`, with time complexity of $O(n^3)$ steps. The basic idea of transformer in [47] is to use a locking mechanism and mutual exclusion to make sure that the execution of the algorithm `MWPIDS` (in our case) is based upon correct distance-2 information; a detailed exposition of the transformer can be found in [47].

To use the transformer in [47] convert algorithm `MWPIDS`, algorithm `MWPIDS1` requires that each node i, $1 \leq i \leq n$, maintain the following extra variables:

- A nonnegative integer variable $\boldsymbol{c_i}$ to keep track of the total amount of influence from node i's $\mathcal{S}$-neighbors, i.e., $c_i = \sum_{j \in N(i) \cap \mathcal{S}} w_{j,i}$, in any system state.
- A pointer $\boldsymbol{grant_i}$ (which may be null) that points to a node $j \in N[i]$, indicated by $grant_i = j$; we say node i *requests* a lock iff $grant_i = i$; and it is *locked* iff $grant_j = i$ for each $j \in N[i]$.
- A boolean flag $\boldsymbol{wtr_i}$; node i sets this bit to express its intent to request the lock.
- A boolean flag $\boldsymbol{wte_i}$; node i sets this bit to express its intent to execute algorithm `MWPIDS`.

Definition 3.2.2 *In any system state,* `minRequest` *of a node* i, $1 \leq i \leq n$, *is defined to be the smallest ID node among the nodes in* $N[i]$ *who requests a lock, i.e.,*

$$\texttt{minRequest}_i = \min\{j | j \in N[i] \wedge grant_j = j\}, \textit{where } \min\{\} = \textit{null}$$

In the following two definitions, a few predicates are defined to facilitate the development of algorithm `MWPIDS1`.

Definition 3.2.3 *In any system state, for a node* i,

1. *A Boolean predicate* $\texttt{needEnter}_i = 1$ *iff* $i \notin \mathcal{S}$ *and there exists at least one illegal node* $j \in N(i)$, *as evidenced by the content of the variable* c_j *on* $j \in N(i)$*:*

$$\texttt{needEnter}_i \stackrel{\text{def}}{\equiv} (s_i = 0) \wedge (\exists j \in N(i) : c_j < t_j)$$

2. *A Boolean predicate* $\texttt{needLeave}_i = 1$ *iff* $i \in \mathcal{S}$ *and each* $j \in N(i)$ *has* $c_j \geq t_j + w_{i,j}$*:*

$$\texttt{needLeave}_i \stackrel{\text{def}}{\equiv} (s_i = 1) \wedge (\forall j \in N(i) : c_j \geq t_j + w_{i,j})$$

Definition 3.2.4 *In any system state, for a node* i,

1. *A Boolean predicate* $\texttt{consistent}_i$ *is true (node* i *is called* ***consistent****), iff* c_i *is correct, i.e.,*

$$\texttt{consistent}_i \stackrel{\text{def}}{\equiv} c_i = \sum_{j \in N(i) \cap \mathcal{S}} w_{j,i}$$

2. *A Boolean predicate* $\texttt{idle}_i$ *is true (node* i *is called* ***idle****), iff* i *is consistent and each node* j *in the closed neighborhood of* i *has* $grant_j = null$*, i.e.,*

$$\texttt{idle}_i \stackrel{\text{def}}{\equiv} \texttt{consistent}_i \wedge (\forall j \in N[i] : grant_j = null)$$

3. *A Boolean predicate* $\texttt{approved}_i$ *is true (node* i *is called* ***approved****), iff* i *is consistent and locked, i.e.,*

$$\texttt{approved}_i \stackrel{\text{def}}{\equiv} \texttt{consistent}_i \wedge (\forall j \in N[i] : grant_j = i)$$

4. *A Boolean predicate* $\texttt{MWPIDS-enabled}_i$ *is true (node* i *is called* ***MWPIDS-enabled****), iff* i *needs either enter or leave* $\mathcal{S}$*, i.e.,*

$$\texttt{MWPIDS-enabled}_i \stackrel{\text{def}}{\equiv} \texttt{needEnter}_i \vee \texttt{needLeave}_i$$

The complete pseudo code of algorithm `MWPIDS1` is shown in Figure 3.2.

R1 : **if** $\neg$**consistent**$_i$
then $c_i \leftarrow \sum_{j \in N(i) \cap \mathcal{S}} w_{j,i}$; **[Update]**

R2 : **if** **consistent**$_i \wedge (\neg$**MWPIDS-enabled**$_i) \wedge (wtr_i = 1)$
then $wtr_i \leftarrow 0$; **[UnlockR]**

R3 : **if** **consistent**$_i \wedge (wte_i = 1) \wedge (\exists j \in N[i] : grant_j \neq i)$
then $wte_i \leftarrow 0$; **[UnlockE]**

R4 : **if** **idle**$_i \wedge$**MWPIDS-enabled**$_i \wedge (wtr_i = 1) \wedge (\forall j \in N(i) : j > i \vee wtr_j = 0)$
then $\{grant_i = i; wtr_i \leftarrow 0;\}$ **[Request]**

R5 : **if** **consistent**$_i \wedge (grant_i \neq$ **minRequest**$_i) \wedge (grant_i = null \vee wte_i = 0)$
then $grant_i \leftarrow$ **minRequest**$_i$; **[Grant]**

R6 : **if** **idle**$_i \wedge$**MWPIDS-enabled**$_i \wedge (wtr_i = 0)$
then $wtr_i \leftarrow 1$; **[LockR]**

R7 : **if** **approved**$_i \wedge$**MWPIDS-enabled**$_i \wedge (wte_i = 0)$
then $wte_i \leftarrow 1$; **[LockE]**

R8 : **if** **approved**$_i \wedge \neg$**MWPIDS-enabled**$_i$
then $\{wte_i \leftarrow 0; wtr_i \leftarrow 0; grant_i \leftarrow null\}$ **[Reset]**

R9 : **if** **approved**$_i \wedge$**MWPIDS-enabled**$_i \wedge (wte_i = 1)$
then { **execute Algorithm MWPIDS**; $wte_i \leftarrow 0; wtr_i \leftarrow 0; grant_i \leftarrow null$; } **[Execute]**

Figure 3.2: Algorithm `MWPIDS1` on node i, $1 \leq i \leq n$

Theorem 3.2.4 *[47] Any self-stabilizing algorithm using the distance-2 model and a central scheduler can be converted to a distance-1 model algorithm using a distributed scheduler at an increased*

$O(n^2)$ *computational cost.*

Theorem 3.2.5 *Algorithm* `MWPIDS1` *generates a minimal weighted positive influence dominating set and terminates in* $O(n^3)$ *steps using distance-1 model and a distributed scheduler.*

Proof: This follows from Theorems 3.2.3 and 3.2.4. □

3.3 An Example Execution of Algorithm `PIDS` [51]

The details of algorithm `PIDS` can be found in [51]. Each node i in algorithm `PIDS` maintains two variables: a boolean flag $\boldsymbol{s_i}$ and a set of pointers $\boldsymbol{p_i} \subseteq N(i)$. In any system state $\mathcal{S}$ is the current set of nodes with $s_i = 1$. Authors in [51] proved that algorithm `PIDS` stabilizes in at most $O(n^2)$ steps under a central scheduler. However, the following example shows that the time complexity of the algorithm `PIDS` is $O(2^{n+1})$.

An Example

Given a complete graph $G = (V, E)$ with n nodes; (a) each edge (v_i, v_j) in E is replaced by two new edges, (v_i, x_{ij}) and (x_{ij}, v_j), and (b) a leaf-node l_i is added to be incident to each node v_i in V; thus a new graph $G' = (V', E')$ with $(n^2 + 3n)/2$ nodes is constructed. Assume nodes $v_1, ..., v_n$ have the largest IDs, while nodes $l_1, ..., l_n$ have the smallest ones; we number the IDs of nodes $v_1, ..., v_n$ in the increasing order. Initially all flags are clear and all points are null; node l_n is selected to move in the first step such that it points to v_n; then we construct a sequence of execution recursively by considering node v_n first as shown in Figure 3.3:

```
/*Consider node v_i*/
{ for (j = i − 1; j ≥ 1; j = j − 1)                          [1]
    then { select x_ij (x_ij points to v_j);                 [2]
           consider node v_j (recursively);                  [3]
  select v_i (v_i enters S);                                 [4]
  for (j = i − 1; j ≥ 1; j = j − 1)                          [5]
    then { select x_{i,j} (x_ij points to null);             [6]
           select v_j (v_j exits S);                         [7]
```

Figure 3.3: An execution sequence of algorithm `PIDS`

C_i is used to denotes the number of moves during the process of considering node v_i,

$1 \leq i \leq n$. Lines 1-3 need $\sum_{1 \leq j < i}(C_j + 1)$ moves; Line 4 needs 1 move; and Lines 5-7 need $2(i-1)$ moves; in total we have:

$$C_i = \begin{cases} \sum_{1 \leq j < i} C_j + 3i - 2 & \text{if } 1 < i \leq n \\ 1 & \text{otherwise} \end{cases} \tag{3.1}$$

Theorem 3.3.1 *The number of moves in the execution sequence generated in Figure 3.3 is exponential in terms of n for graph G'.*

Proof: Consider $n > 1$: according to Equation 3.1, we have:

$$\begin{aligned} C_n &= \sum_{1 \leq k < n} C_k + 3n - 2 \\ &= C_{n-1} + \left(\sum_{1 \leq k < n-1} C_k + 3(n-1) - 2 \right) + 3 \\ &= 2C_{n-1} + 3 \\ &= 2^{n-1} C_1 + 3 \times \sum_{0 \leq k \leq n-2} 2^k \\ &= 2^{n-1} + 3(2^{n-1} - 1) \\ &= 2^{n+1} - 3 \end{aligned} \tag{3.2}$$

Thus, the theorem holds. □

Chapter 4

Self-Stabilizing Algorithms with Safe Convergence

In a traditional self-stabilizing algorithm, the desired global property (the relevant service in the system) is not guaranteed during the convergence interval; the concept of *safe convergence* was introduced to limit this inconvenience to a minimum possible. A self-stabilizing algorithm has the safe convergence property iff it first converges to a safe (feasible but sub-optimal) state in constant time, and then converges to a legitimate (optimal) state without breaking safety in the process [32]. Safe convergence property is especially attractive since it provides a measure of safety (desired service at a sub-optimal level) during the convergence interval until the optimal legitimate state is reached.

This chapter is interested in minimal (f, g)-Alliance (Definition 1.3.6) and maximal k-packing (Definition 1.3.7) in the network graph. Assume each node has a unique ID and a synchronous scheduler, a self-stabilizing algorithm with safe convergence is proposed to compute the minimal (f, g)-alliance of an arbitrary network graph. Starting from an arbitrary initial state, the proposed algorithm quickly converges to a (f, g)-alliance (a *safe* state) in three steps, and then terminates in a minimal one (the *legitimate* state) in $O(n)$ steps without breaking safety rule during the state transitions, where n is the number of nodes in the network graph. Then, another self-stabilizing algorithm with safe convergence is proposed to compute the maximal 2-packing of an arbitrary network graph; starting from an arbitrary state, the proposed algorithm first converges to

a 2-packing (a *safe* state) in three steps, and then converges to a maximal one (the *legitimate* state) in $O(n)$ steps without breaking safety rule during the stabilization interval. Both algorithms have space complexity of $O(\log n)$ bits at each node. At last, the technique is generalized to design a self-stabilizing algorithm for maximal k-packing, $k \geq 2$, with safe convergence that stabilizes in $O(kn^2)$ steps under synchronous scheduler; the proposed algorithm has space complexity of $O(kn \log n)$ bits at each node.

4.1 Model and Terminology

Execution Model: The proposed algorithms assume that each node has a unique identifier 1 through n. The execution of the algorithm at each node is managed by a synchronous scheduler, that selects all privileged nodes in a system state to move synchronously and atomically in each round (note that such a round is different from the concept of a round used in fair central or distributed schedulers; such a round is also called a synchronous step [32, 34, 36, 15, 42]). State-reading model is used as opposed to read/write atomicity model [18]. A global system state, the union of the local states of all nodes, is denoted by Σ_i, $i = 0, 1, 2, \cdots$, where Σ_0 denotes the initial arbitrary state and Σ_r denotes the system state after the r-th step of the algorithm, $r = 1, 2, \cdots$; r-th step executes on Σ_{r-1} to generate Σ_r.

A node is privileged in a given system state iff it is enabled to move by at least one rule of the algorithm. The algorithm *terminates* in a system state when no node is privileged. The algorithm assumes a *shared-memory model* and each node knows only its own state and the local states of its immediate neighbors (distance-one model) as is customary in the most self-stabilizing algorithms.

4.2 Minimal (f, g)-Alliance with Safe Convergence

In this section, a new self-stabilizing algorithm with safe convergence to compute the minimal (f, g)-alliance of an arbitrary network graph is presented. Starting from an arbitrary initial state, the proposed algorithm quickly converges to a (f, g)-alliance (a *safe* state) in three steps, and then terminates in a minimal one (the *legitimate* state) in $O(n)$ steps without breaking safety rule during the state transitions, where n is the number of nodes in the network graph.

4.2.1 Algorithm MFGASC

In the proposed self-stabilizing minimal (f, g)-alliance algorithm with safe convergence (called algorithm MFGASC), each node $i, 1 \leq i \leq n$, maintains the following variables:

- A boolean flag $\boldsymbol{s_i}$; at any time (system state) $\mathcal{S}$ is the current set of nodes with $s_i = 1$.
- A nonnegative integer variable $\boldsymbol{c_i}$ to count the number of $\mathcal{S}$-neighbors of node i at any given system state.
- A pointer $\boldsymbol{p_i}$ (which may be null) that points to a node $j \in N[i]$, indicated by $p_i = j$. If, in a system state, $p_i = i$ for a node i, we say node i has a **self-pointer**.
- A boolean flag $\boldsymbol{d_i}$; node i sets this bit to delay some activity by one step only.

Definition 4.2.1 *In a system state, a node i is called **consistent** if either (a) node i is out of $\mathcal{S}$ and has $|N(i) \cap \mathcal{S}| \geq f_i$, or (b) node i is in $\mathcal{S}$ and has $|N(i) \cap \mathcal{S}| \geq g_i$.*

Definition 4.2.2 *A system state is **safe** if $\mathcal{S} = \{i \in V : s_i = 1\}$ denotes a (f, g)-alliance, i.e., each node in V is consistent. A system state is **legitimate** if $\mathcal{S}$ denotes a minimal (f, g)-alliance.*

Definition 4.2.3 *In any system state, $minSP_i$ of a node i, $1 \leq i \leq n$, is defined to be the smallest ID node among the nodes in $N[i]$ with self-pointer, i.e.,*

$$minSP_i = \min\{j | j \in N[i] \wedge p_j = j\}, \textit{where } \min\{\} = null$$

The objective of algorithm MFGASC is to quickly converge to a safe state and thereafter to transition through safe states to reach the legitimate state to obtain the minimal (f, g)-alliance. The algorithm assumes a synchronous scheduler where at any step all privileged nodes are selected to move. The underlying approach is as follows:

- (**Enter** $\mathcal{S}$) A node $i \notin \mathcal{S}$ enters $\mathcal{S}$ iff some neighbor(s) $j \in N(i)$ is inconsistent.
- (**Exit** $\mathcal{S}$) A node $i \in \mathcal{S}$ exits $\mathcal{S}$ iff it can ensure each node in $N[i]$ remains consistent after the current step.

In the following two definitions, a few predicates are defined to facilitate the stepwise development of the proposed algorithm.

Definition 4.2.4 *For a node i, a Boolean predicate* $\texttt{nowEnter}_i = 1$ *iff* $i \notin \mathcal{S}$ *and some neighbor j of i is inconsistent:*

$$\texttt{nowEnter}_i \stackrel{\text{def}}{\equiv} (s_i = 0) \wedge \Big((\exists j \in N(i) - \mathcal{S} : c_j < f_j) \vee (\exists j \in N(i) \cap \mathcal{S} : c_j < g_j) \Big)$$

Definition 4.2.5 *For a node i, a Boolean predicate* $\texttt{needExit}_i = 1$ *iff* $i \in \mathcal{S}$ *has* $c_i \geq f_i$ *and each neighbor* $j \in \mathcal{S}$ *has* $c_j > f_i$*, and each neighbor* $j \notin \mathcal{S}$ *has* $c_j > g_i$*:*

$$\texttt{needExit}_i \stackrel{\text{def}}{\equiv} (s_i = 1) \wedge (c_i \geq f_i) \wedge \Big((\forall j \in N(i) - \mathcal{S} : c_j > f_j) \wedge (\forall j \in N(i) \cap \mathcal{S} : c_j > g_j) \Big)$$

Remark 4.2.1 *In any system state, it is possible that c_j's are erroneous. Under synchronous scheduler:*

1. *If node i has* $\texttt{nowEnter}_i = 1$*, it makes a enter move (changing s bit to 1); Note that the enter move of i may decrease but never increase the number of inconsistent nodes in $N(i)$.*
2. *If node i has only* $\texttt{needLeave}_i = 1$*, it can not make an exit move (changing s bit to 0), otherwise the exit move of i may result in inconsistent nodes in $N(i)$.*

The algorithm requires that after a node i exits $\mathcal{S}$ in a step, each node in $N[i]$ remains consistent at the end of current step. A locking mechanism (implemented by stored variable p_i) is used such that when a node exits $\mathcal{S}$ in a step, all other nodes in its 2-hop neighborhood are prohibited to exit $\mathcal{S}$. Specifically, when a node i needs to exit $\mathcal{S}$, it first requests a lock by setting self-pointer (i.e., $p_i = i$). A node i is locked or gets the lock iff all nodes in $N[i]$ point to i. The neighbor j of i grants the lock by updating its pointer to i, i.e., $p_j = i$. It is possible that two adjacent nodes request locks simultaneously. In order to break the tie, the node grants the lock (by updating its pointer) to the smallest ID neighbor with self-pointer.

Definition 4.2.6 *To implement* ***locking mechanism****, two more Boolean predicates* $\texttt{requestLock}_i$ *and* $\texttt{locked}_i$ *on node i are defined as:*

$$\texttt{requestLock}_i \stackrel{\text{def}}{\equiv} \texttt{needExit}_i \wedge (\forall j \in N[i] : p_j = null)$$

$$\texttt{locked}_i \stackrel{\text{def}}{\equiv} \forall j \in N[i] : p_j = i$$

Remark 4.2.2 *In a system state, if node i is locked, no node in $N^2(i)$ can be locked in the same state.*

After a node i becomes locked, it sets d bit to delay its exit move by one step only. It should be noted that in delayed step: each node $j \in N[i]$ must have no locked neighbor except i (Remark 4.2.2) and hence no neighbor exiting $\mathcal{S}$; each neighbor $j \in N[i]$ gets chance to update c_j, such that $c_j \leq |N(j) \cap \mathcal{S}|$ are guaranteed when node i exits (it is possible that some neighbor of $j \in N[i]$ enters $\mathcal{S}$ in step r, hence the inequality). After the delayed step, node i is guaranteed to be safe to exit (i.e., after node i exits $\mathcal{S}$, each node in $N[i]$ remains consistent).

Definition 4.2.7 *In any system state, a node i can **exit** $\mathcal{S}$ iff the Boolean predicate* $\texttt{nowExit}_i = 1$ *where*

$$\texttt{nowExit}_i \stackrel{\text{def}}{\equiv} (d_i = 1) \wedge \texttt{needExit}_i \wedge \texttt{locked}_i$$

Remark 4.2.3 *In any system state, if a node i is enabled to exit (*$\texttt{nowExit}_i = 1$*), no node in $N^2(i)$ is enabled to exit in the same state since no node in $N^2(i)$ can be locked in that state (Remark 4.2.2).*

Definition 4.2.8 *For the sake of brevity, in any system state, four more Boolean predicates on node i are defined as:*

$$\texttt{updateC}_i \stackrel{\text{def}}{\equiv} c_i \neq |N(i) \cap \mathcal{S}|$$

$$\texttt{updateP}_i \stackrel{\text{def}}{\equiv} p_i \neq \mathit{minSP}_i$$

$$\texttt{clearD}_i \stackrel{\text{def}}{\equiv} (d_i = 1) \wedge \neg(\texttt{needExit}_i \wedge \texttt{locked}_i)$$

$$\texttt{requestDelay}_i \stackrel{\text{def}}{\equiv} (d_i = 0) \wedge \texttt{needExit}_i \wedge \texttt{locked}_i$$

$$\texttt{releaseLock}_i \stackrel{\text{def}}{\equiv} \neg\texttt{needExit}_i \wedge (p_i = i) \wedge (\mathit{minSP} = i)$$

The complete pseudo code of algorithm `MFGASC` is shown in Figure 4.1. A few simple characteristics of the algorithm are highlighted in the following remark.

Remark 4.2.4 *In a given step r, $r \geq 1$, of execution:*

1. *If node i has incorrect c_i in Σ_{r-1}, it must update c_i in step r.*
2. *For a node i,* $\texttt{nowEnter}_i$, $\texttt{requestDelay}_i$ *and* $\texttt{nowExit}_i$ *are pairwise mutual exclusive;* $\texttt{requestLock}_i$, $\texttt{releaseLock}_i$ *and* $\texttt{updateP}_i$ *are pairwise mutual exclusive.*

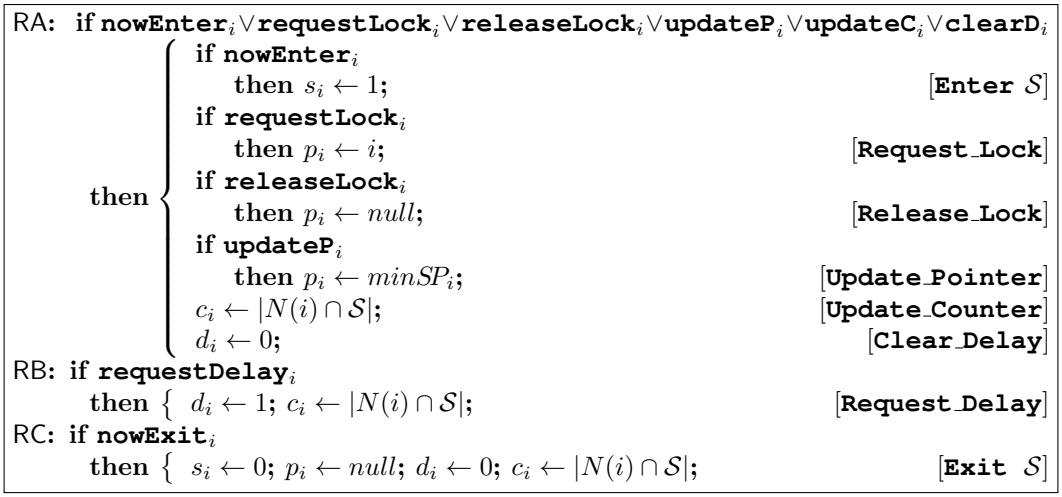

Figure 4.1: Algorithm MFGASC on node i, $1 \leq i \leq n$

3. *The membership of node i is changed only by rules* RA *(Enter $\mathcal{S}$ move) and* RC *(Exit $\mathcal{S}$ move). If a node i is privileged to make Enter/Exit $\mathcal{S}$ move, it must enter/exit $\mathcal{S}$ successfully under synchronous scheduler (see part(2)).*

4. *If node i enters $\mathcal{S}$, its neighboring nodes can concurrently enter $\mathcal{S}$ if they are eligible to do so; If node i exits $\mathcal{S}$, no node $j \in N^2(i)$ can concurrently exit $\mathcal{S}$ in the same step (Remark 4.2.3).*

5. *A node i can acquire a self-pointer ($p_i = i$) only by making Reqeust_Lock move in rule* RA *when it needs to exit $\mathcal{S}$ to minimize $|\mathcal{S}|$ and all its neighbors have null pointers.* ***Note****: a node cannot acquire a self-pointer by making Update_Pointer move in rule* RA.

6. *A node i releases its self-pointer when it does not need to exit $\mathcal{S}$ (*$\texttt{needExit}_i = 0$*) by making Release_Lock move, or when it has at least one smaller ID neighbor with self-pointer by making Update_Pointer move.*

7. *After a node i with* $\texttt{needExit}_i = 1$ *becomes locked, i.e.,* $\texttt{locked}_i = 1$*, it delays its exit move by one step only by making Request_Delay move (setting $d_i = 1$) such that its neighbors have time to correct their c-variables.*

8. *If $d_i = 1$ in Σ_{r-1}, then node i will clear delay by making either Exit $\mathcal{S}$ or Clear_Delay move such that $d_i = 0$ in Σ_r.*

Definition 4.2.9 *In any system state $\Sigma_r, r \geq 0$:*

*(a) A node i is **privileged** if it is enabled by any of the rules of the algorithm.*

*(b) The execution of the algorithm **terminates** when no node is privileged.*

4.2.2 Correctness & Convergence

In this section, we first prove that $\mathcal{S}$ is a minimal (f, g)-alliance when algorithm `MFGASC` terminates; then we show the algorithm has safe convergence property. Starting from an arbitrary state, the proposed algorithm first converges to a (f, g)-alliance (a safe state) in 3 steps, and then stabilizes to a minimal (f, g)-alliance (the legitimate state) in $O(n)$ steps without breaking safety, where n is the number of nodes.

Lemma 4.2.1 *If algorithm `MFGASC` terminates, then for each node $i \in V$*

(a) c_i is correct, i.e., $c_i = |N(i) \cap \mathcal{S}|$.

(b) $d_i = 0$.

(c) $p_i = null$.

(d) $\texttt{nowEnter}_i = 0$ and $\texttt{needExit}_i = 0$.

Proof: (a) This lemma immediately follows from the fact that no node is privileged by the rule RA at the termination of the algorithm.

(b) Assume, by contradiction, there exists some node(s) j with $d_j = 1$. Node j must have $\texttt{needExit}_j = 1$ and $\texttt{locked}_j = 1$ (otherwise node j is privileged by rule RA to make Clear_Delay move). Thus, node j is privileged by rule RC, a contradiction.

(c) If no node j has self-pointer, then $minSP_i = null$ for all $i \in V$ and hence the lemma holds since $p_i = minSP_i$ (otherwise node i is privileged by rule RA to make Update_Pointer move). So the key point here is to show that there is no node with self-pointer. Assume, by contradiction, there exists some node(s) with self-pointer. Consider the node with minimum ID from among those nodes, say node j; $minSP_k = j$ for each node $k \in N[j]$. Also each node $k \in N[j]$ must have $p_k = minSP_k = j$ (otherwise node k would be privileged by the rule RA to make Update_Pointer move). Thus, node j is locked (i.e., $\texttt{locked}_j = 1$). Also, node j must have $\texttt{needExit}_j = 1$ (otherwise node j is privileged by rule RA to make Release_Lock move). Thus, we get node j is privileged by rule RB to make Request_Delay move (by part(b)), a contradiction.

(d) No node i is privileged by rule RA to make Enter $\mathcal{S}$ move and Request_lock move; the claim follows from parts (a) and (c). □

Theorem 4.2.1 *Starting from an arbitrary system state, if algorithm* `MFGASC` *terminates using synchronous scheduler, then $\mathcal{S}$ is a minimal (f, g)-alliance.*

Proof: First, we show $\mathcal{S}$ is a (f, g)-alliance. Assume, by contradiction, $\mathcal{S}$ is not a (f, g)-alliance, i.e., there is at least one inconsistent node i (Definition 4.2.2). Thus $\texttt{nowEnter}_j = 1$ for each neighbor $j \in N(i) - \mathcal{S}$; node j is privileged by the rule RA (Enter $\mathcal{S}$). A contradiction, thus $\mathcal{S}$ is a (f, g)-alliance.

Next, we claim $\mathcal{S}$ is minimal. Assume otherwise, i.e., there exists a node $i \in \mathcal{S}$ such that $\mathcal{S} - \{i\}$ is a (f, g)-alliance. Since $i \in \mathcal{S}$ (i.e., $s_i = 1$), and $\texttt{needExit}_i = 0$ (Lemma 4.2.1(d)), either (1) node i has $c_i < f_i$, or (2) some neighbor(s) $j \in N(i) - \mathcal{S}$ has $c_j \leq f_j$, or (3) some neighbor(s) $j \in N(i) \cap \mathcal{S}$ has $c_j \leq g_j$. Thus, the removal of i from $\mathcal{S}$ would make either node i or at least one of its neighbors inconsistent, i.e., $\mathcal{S} - \{i\}$ is not a (f, g)-alliance, a contradiction. □

Lemma 4.2.2 *In any system state Σ_r, $r \geq 1$, if a node i is enabled by the rule* RC *to exit $\mathcal{S}$, each node $j \in N[i]$ must have $c_j \leq |N(j) \cap \mathcal{S}|$.*

Proof: In the system state Σ_{r-1}, node i must have had $d_i = 0$, $\texttt{needExit}_i = 1$ and $\texttt{locked}_i = 1$; otherwise it is impossible for node i to have $d_i = 1$ in Σ_r (Remark 4.2.4.8 and Request_Delay move). Since $\texttt{locked}_i = 1$ in Σ_{r-1}, no node $j \in N^2(i)$ was locked (Remark 4.2.2) and hence exited $\mathcal{S}$ in step r. Coupled with the fact that each node $j \in N[i]$ corrected its c_j in step r (Remark 4.2.4.1), thus, in Σ_r, each node $j \in N[i]$ must have $c_j \leq |N(j) \cap \mathcal{S}|$. **Note:** it is possible that some neighbor of node j enters $\mathcal{S}$ in step r (hence the inequality). □

Lemma 4.2.3 *In step $r \geq 2$, if a node i exits $\mathcal{S}$ (by executing rule* RC*), each node $j \in N[i]$ remains consistent.*

Proof: This lemma immediately follows from Lemma 4.2.2 and Definition 4.2.7. □

Lemma 4.2.4 *In system state Σ_2, (a) if node $i \in \mathcal{S}$ has $|N(i) \cap \mathcal{S}| < g_i$ (i.e., node i is inconsistent), then c_i must be less than g_i; (b) if node $i \notin \mathcal{S}$ has $|N(i) \cap \mathcal{S}| < f_i$ (i.e., node i is inconsistent), then c_i must be less than f_i*

Proof: Consider the case (a), we observe that: (1) no node $j \in N[i] \cap \mathcal{S}$ exited $\mathcal{S}$ in step 2 (otherwise node i should be consistent in Σ_2 by Lemma 4.2.3); (2) $c_i \geq g_i$ in Σ_1 (otherwise node i should be consistent in Σ_2 since all the nodes $j \in N[i] - \mathcal{S}$ would enter $\mathcal{S}$ in step 2); thus no node $j \in N[i] - \mathcal{S}$ entered $\mathcal{S}$ in step 2. We therefore conclude that no node $j \in N[i]$ left/entered $\mathcal{S}$ in step 2. Coupled with the fact that $|N(i) \cap \mathcal{S}| < g_i$ in Σ_2, we observe node i must have had $|N(i) \cap \mathcal{S}| < g_i$ in Σ_1; further node i must have $c_i = |N(i) \cap \mathcal{S}| < g_i$ in Σ_2 by executing rule RA to correct c_i in step 2 (Remark 4.2.4.1).

The argument for (b) is similar to the one for (a); we omit the details. □

Theorem 4.2.2 *Starting from any initial illegitimate state, algorithm MFGASC converges to a safe state ($\mathcal{S}$ denotes a (f, g)-alliance, i.e., each node i is consistent) after 3 steps.*

Proof: We show that each node i is consistent (Definition 4.2.1) after 3 steps of execution; we consider two cases:

(a) Consider any node $i \in \mathcal{S}$ with $|N(i) \cap \mathcal{S}| < g_i$ in Σ_2: $c_i < g_i$ by Lemma 4.2.4(a), each node $j \in N[i] \cap \mathcal{S}$ has $\texttt{needExit}_j = 0$ and each node $j \in N[i] - \mathcal{S}$ has $\texttt{nowEnter}_j = 1$. Thus, in step 3 no node $j \in N[i] \cap \mathcal{S}$ exits $\mathcal{S}$ but each node $j \in N[i] - \mathcal{S}$ must enter $\mathcal{S}$ by executing the rule RA (Remark 4.2.4.3). Hence node i will be consistent ($|N[i] \cap \mathcal{S}| \geq g_i$) in Σ_3.

(b) Consider any node $i \notin \mathcal{S}$ with $|N(i) \cap \mathcal{S}| < f_i$ in Σ_2: $c_i < f_i$ by Lemma 4.2.4(b), each node $j \in N[i] \cap \mathcal{S}$ has $\texttt{needExit}_j = 0$ and each node $j \in N[i] - \mathcal{S}$ has $\texttt{nowEnter}_j = 1$. Thus, in step 3 no node $j \in N[i] \cap \mathcal{S}$ exits $\mathcal{S}$ but each node $j \in N[i] - \mathcal{S}$ enters $\mathcal{S}$ by executing the rule RA(Remark 4.2.4.3). Hence node i will be consistent ($|N[i] \cap \mathcal{S}| \geq f_i$) in Σ_3.

(c) Consider any other nodes i in Σ_2: If any $\mathcal{S}$ node in the closed neighborhood of node i exits $\mathcal{S}$ in step 3, node i remains consistent in Σ_3 (Lemma 4.2.3). □

Theorem 4.2.3 *After step 3, algorithm MFGASC maintains safety in all subsequent steps before converging to a legitimate state.*

Proof: After step 3, we reach a safe state. In any safe state, any node i in V is consistent (Definition 4.2.1); thus no node enters $\mathcal{S}$ (Definition 4.2.4). If any neighbor of i exits $\mathcal{S}$, node i will remain consistent in the next state by Lemma 4.2.3, thus we reach another safe state. □

Lemma 4.2.5 *Starting from a safe state Σ_r, $r \geq 3$, no node will ever make Enter $\mathcal{S}$ move in subsequent steps.*

Proof: In a safe state, each node is consistent, i.e., $\texttt{nowEnter}_i$ (Definition 4.2.4) is false for each node $i \in V$; thus, no node i can make Enter $\mathcal{S}$ move by executing rule RA in the current step. The algorithm MFGASC always in the safe state after step 3 (Theorem 4.2.3), thus each node always remains consistent after step 3. The lemma holds. □

Lemma 4.2.6 *In any safe state Σ_r, $r \geq 4$, rules* RA, RB *and* RC *are pairwise mutual exclusive.*

Proof: It is easy to show requestDelay and nowExit are pairwise mutual exclusive with each of nowEnter, requestLock, releaseLock, updateP and clearD, we here omit the details. Coupled with the fact that requestDelay and nowExit are pairwise mutual exclusive. Thus, to prove the lemma, it suffices that show that requestDelay and nowExit are pairwise mutual exclusive with updateC:

For any node i, if $\texttt{requestDelay}_i = 1$ or $\texttt{nowExit}_i = 1$ in Σ_r, no neighbor $j \in N[i]$ exited $\mathcal{S}$ in the step r (otherwise node i cannot have self-pointer in Σ_r). Coupled with Lemma 4.2.5 and Remark 4.2.4.1, c_i must be correct in Σ_r. Thus, $\texttt{requestDelay}_i$ and $\texttt{updateC}_i$ are pairwise mutual exclusive. □

Lemma 4.2.7 *In any system state Σ_r where $r \geq 1$, two adjacent nodes i and j have self-pointers (i.e., $p_i = i$ and $p_j = j$), then (a) the larger ID node will lose the self-pointer (i.e., if say $i < j$, $p_j \neq j$) in the next steps; (b) nodes i and j must have concurrently acquired the self-pointers in step r.*

Proof: (a) In the next step node j will make Update_Pointer move in rule RA. (b) If $p_i = i$ but $p_j \neq j$, then node j cannot make Request_Lock move to get the self-pointer in the next state (Definition 4.2.6). □

Lemma 4.2.8 *In a safe state Σ_r, $r \geq 3$, for two adjacent nodes i and j, if $p_i = i$ and $p_j = j$, then* $\texttt{needExit}_i = \texttt{needExit}_j = 1$.

Proof: It follows from Lemma 4.2.7 that nodes i and j had concurrently made Request_Lock move to get self-pointers in step r. In Σ_{r-1}, $\texttt{needExit}_i = 1$, $\texttt{needExit}_j = 1$, $p_i = p_j = null$, and for each $k \in N(i) \cup N(j)$, $p_k = null$ (nodes i and j are enabled for rule RA to make Request_Lock move; Definition 4.2.5). We argue that:

(a) Node i can not exit $\mathcal{S}$ (execute rule RC) in step r since $\texttt{locked}_i = 0$ in Σ_{r-1} ($p_i = null$).

(b) Any node $k \in N(i)$ cannot exit $\mathcal{S}$ (execute rule RC) in step r since $\texttt{locked}_k = 0$ in Σ_{r-1} ($p_k = null$).

(c) Any neighbor k' of $k \in N(i)$ cannot exit $\mathcal{S}$ (execute rule RC) in step r since $\texttt{locked}_{k'} = 0$ in Σ_{r-1} ($p_k = null$).

Thus, $\texttt{needExit}_i$ remains 1 in Σ_r (by similar reasoning, $\texttt{needExit}_j$ remains 1 in Σ_r). □

Definition 4.2.10 *In any system state,*

(a) We define an island $\mathcal{I}$ to be a maximal *set of nodes $\{i \in V | \texttt{needExit}_i = 1 \wedge p_i = i\}$ such that the subgraph of G induced by the set $\mathcal{I}$ is connected.*

(b) We use $\boldsymbol{\alpha}$ to denote the number of islands and $\boldsymbol{\beta}$ to denote the number of nodes i with $\texttt{needExit}_i = 1$.

Remark 4.2.5 *In any system state:*

1. *An island may consist of a single or multiple nodes; a node i with $\texttt{needExit}_i = 1$ and $p_i \neq i$ is not a member of any island.*
2. *For a node i in an island of size ≥ 2, $\texttt{locked}_i = 0$ since it has a neighbor j with $p_j = j \neq i$ (Definition 4.2.6).*
3. *$\alpha \leq \beta$; $\alpha < n$; $\beta \leq n$;*
4. *After step 3, β is non increasing in subsequent steps (Definition 4.2.5 and Lemma 4.2.5).*
5. *When algorithm MFGASC terminates, $\alpha = \beta = 0$.*

Lemma 4.2.9 *If a node i exits $\mathcal{S}$ (by executing rule* RC*) in a step, node i constitutes a single node island at the beginning of the step.*

Proof: Node i exits $\mathcal{S}$; thus $\texttt{needExit}_i = 1$ and $\texttt{locked}_i = 1$ (rule RC). Since $p_i = i$, node i belongs to an island (Definition 4.2.10); node i does not belong to an island of size ≥ 2 (Remark 4.2.5.2). □

Lemma 4.2.10 *In any step r, $r \geq 4$ (starting from a safe state Σ_{r-1}), (a) α can not decrease if β remains constant; (b) α decreases at least by 1 and at most by ℓ, if β decreases by ℓ ($1 \leq \ell \leq \beta$).*

Proof: (a) If β remains constant, no node i changes $\texttt{needExit}_i$ from 1 to 0. (1) Any island $\mathcal{I}$ cannot disappear since the smallest ID node in $\mathcal{I}$, say node i, cannot change its pointer in step r ($p_i = i = minSP_i$ in Σ_{r-1} [no neighbor j of i with $\texttt{needExit}_j = 0$ has a self-pointer by Lemma 4.2.8 and node i does not have any island node neighbor with a smaller ID]). (2) Two islands cannot merge into one: consider any 2 islands $\mathcal{I}_1$ and $\mathcal{I}_2$; since $\mathcal{I}_1 \cup \mathcal{I}_2 = \emptyset$, for the two islands to merge, there must be a node $j \in N(\mathcal{I}_1 \cup \mathcal{I}_2)$ such that $\texttt{needExit}_j = 1$ in Σ_{r-1} and node j acquires self-pointer in Σ_r (j becomes an island node in Σ_r) by executing rule RA to make Request_Lock move in step r; this is impossible since j has neighbor(s) with non null pointers in Σ_{r-1} (see Definition 4.2.6).

(b) Starting in a safe state Σ_{r-1}, if β decreases by ℓ in Σ_r, ℓ nodes have changed their needExit bits from 1 to 0. Consider any node i whose $\texttt{needExit}_i$ is changed from 1 to 0. At least one of the three must occur in step r (Definition 4.2.5): (1) node i exits $\mathcal{S}$ by executing rule RC; (2) some neighbor $j \in N(i)$ exits $\mathcal{S}$ by executing rule RC such that $c_i < f_i$; (3) some neighbor(s) of node $j \in N(i) \cap \mathcal{S}$ exits $\mathcal{S}$ by executing rule RC such that $c_j < g_j$; or (4) some neighbor(s) of node $j \in N(i) - \mathcal{S}$ exits $\mathcal{S}$ by executing rule RC such that $c_j < f_j$. If all ℓ nodes change their needExit from 1 to 0 because of (1), then α is decreased by ℓ by Lemma 4.2.9; If some node(s) i changes $\texttt{needExit}_i$ from 1 to 0 because of (2) or (3) or (4), then it is possible that node i does not belong to any island. Although the change of $\texttt{needExit}_i$ on node i causes β to decrease in Σ_r, it does not cause α to decrease (if node i is not an island node in Σ_{r-1}); but, for the possibilities (2) or (3) or (4), at least some other node must exit $\mathcal{S}$ (change s bit to 0) by executing rule RC in the step, thereby causing α to decrease (Lemma 4.2.9). Thus, α decreases by at most ℓ and at least by 1 in Σ_r. □

Lemma 4.2.11 *Starting from a safe state with $\beta \neq 0$,*

(a) either α increases in at most 3 next steps, if β remains constant;

(b) or β decreases in at most 4 next steps.

Proof: Starting from a safe state Σ_r, $r \geq 3$, any node i with $\texttt{needExit}_i = 0$ and $p_i = i$ must either make Update_Pointer or Release_Lock move in step $r+1$ to make $p_i \neq i$ in Σ_{r+1}. Then, Σ_{r+1} does not have a node j with $\texttt{needExit}_j = 0 \wedge p_j = j$ (otherwise, node j had $\texttt{needExit}_j = 1$ in Σ_r and hence β has decreased by at least 1 in one step). There are two possibilities:

(1) There is no island node: In Σ_{r+1}, each node i has $minSP_i = null$ (no node with self-pointer and Definition 4.2.3). Also, since $\beta \neq 0$, there must be a node k with $\texttt{needExit}_k = 1$;

node k, in the worst case, must make Update_Pointer move in step $r+2$ and Request_Lock move in step $r+3$ in that sequence to get $p_k = k$. Thus, there is a new island $\{k\}$, i.e., α has increased in at most 3 steps starting in Σ_r.

(2) There is at least one island node: If there are multiple such island nodes, let i be the node with minimum ID among those. In the worst case, each node $j \in N(i)$ makes Update_Pointer move in step $r+2$ to update their pointers to i; node i becomes locked, i.e., $\texttt{locked}_i = 1$ (Definition 4.2.6), in Σ_{r+2}. Also, if $d_i = 1$ in Σ_{r+1}, node i makes Clear_Delay move in step $r+2$ such that $d_i = 0$ in Σ_{r+2}. Now, there are two possibilities:

(i) At least one $j \in N(i)$ has $minSP_j = k$ in Σ_{r+2} where $k \in N(j)$ (Definition 4.2.3), $p_k = k$, and $k < i$. Node k must have acquired its self-pointer by making Request_Lock move in step $r+2$ and so, $\texttt{needExit}_k = 1 \wedge (\forall k' \in N(k) : p_{k'} = null)$ in Σ_{r+1}, i.e., node k is not connected to any island nodes. Thus, $\{k\}$ is a newly formed single node island in Σ_{r+2}, i.e., α increases in at most 2 steps.

(ii) Each $j \in N(i)$ has $minSP_j = i$ in Σ_{r+2}; in step $r+3$, node i makes Request_Delay move to delay its exit move by one step only. Thus, node i executes rule RC to exit $\mathcal{S}$ in step $r+4$; so $\texttt{needExit}_i = 0$ in Σ_{r+4}, i.e., β decreases in at most 4 steps.

□

Lemma 4.2.12 *After step 3, algorithm MFGASC reaches a safe state with $\alpha = \beta = 0$ in at most $O(n)$ steps under a synchronous scheduler.*

Proof: In a safe state where $\alpha = \beta$, if β decreases by 1, α must decrease by 1 (Lemma 4.2.10); $\beta \leq n$ and β is non-increasing (Remarks 4.2.5.3 and 4.2.5.4). Recall that β decreases by at least 1 in at most 4 steps (Lemma 4.2.11(b)). Thus, from any safe system state with $\alpha = \beta$, the system will be in a safe state with $\alpha = \beta = 0$ in at most $4n$ steps. Also, if β remains constant, α must increase by 1 in at most 3 steps (Lemma 4.2.11(a)); in at most $3n$ steps, α will be equal to β. Thus, the system will be in a safe state with $\alpha = \beta = 0$ in at most $4n + 3n = 7n$ steps. □

Theorem 4.2.4 *Starting in any arbitrary state, the algorithm MFGASC terminates in $O(n)$ steps under a synchronous scheduler.*

Proof: The system reaches a safe state with $\alpha = \beta = 0$ in $7n + 3$ steps in the worst case (Theorem 4.2.2 and Lemma 4.2.12). In the next at most 2 steps all node pointers will be *null* and all d variables will be 0; thus the algorithm terminates. □

4.3 Graph Packing with Safe Convergence

In this section, the first self-stabilizing algorithm with safe convergence to compute the maximal 2-packing of an arbitrary network graph is proposed; starting from an arbitrary state, the proposed algorithm first converges to a 2-packing (a *safe* state) in three steps, and then converges to a maximal one (the *legitimate* state) in $O(n)$ steps without breaking safety rule during the stabilization interval. Space requirement at each node is $O(\log n)$ bits. Then the technique is generalized to design a self-stabilizing algorithm for maximal k-packing, $k \geq 2$, with safe convergence that stabilizes in $O(kn^2)$ steps under synchronous scheduler; the algorithm has space complexity of $O(kn \log n)$ bits at each node.

4.3.1 Maximal 2-packing with Safe Convergence

In the proposed self-stabilizing maximal 2-packing algorithm with safe convergence (called algorithm `M2PSC`), each node $i, 1 \leq i \leq n$, maintains the following variables:

- A boolean flag $\boldsymbol{s_i}$; at any time (system state) $\mathcal{S}$ is the current set of nodes with $s_i = 1$.
- A nonnegative integer variable $\boldsymbol{c_i}$ to count the number of $\mathcal{S}$ nodes in the closed neighborhood of node i, i.e., $c_i = |N[i] \cap \mathcal{S}|$, at any given system state.
- A pointer $\boldsymbol{p_i}$ (which may be null) that points to a node $j \in N[i]$, indicated by $p_i = j$. If, in a system state, $p_i = i$ for a node i, we say node i has a **self-pointer**.
- A boolean flag $\boldsymbol{d_i}$; node i sets this bit to delay some activity by one step only.

Definition 4.3.1

1. *A node i is called **consistent** in a (global) system state if $|N[i] \cap \mathcal{S}| \leq 1$.*
2. *A system state is **safe** if $\mathcal{S} = \{i | i \in V \wedge s_i = 1\}$ denotes a 2-packing, i.e, each node in V is consistent. A system state is **legitimate** if $\mathcal{S}$ denotes a maximal 2-packing.*

3. *In any system state, minSP$_i$ of a node i, $1 \leq i \leq n$, is defined to be the smallest ID node among the nodes in $N[i]$ with self-pointer, i.e., minSP$_i$ $= \min\{j | j \in N[i] \wedge p_j = j\}$, where* $\min\{\} = null$.

Remark 4.3.1 ***(Node Consistency)***

1. *The stored variable c_i at node i is a measure of the local consistency of node i in a system state. If c_i is correct in a system state, i.e., $c_i = |N[i] \cap \mathcal{S}|$, node i is* ***consistent*** *iff $c_i \leq 1$.*

2. *If c_i is not known to be correct in a system state, node i is* ***deemed to be inconsistent*** *iff $c_i \geq 2$.*

The approach underlying the algorithm `M2PSC` is to quickly converge to a safe state, by allowing nodes to exit $\mathcal{S}$ to eliminate all inconsistent nodes in the system state and thereafter to transition through safe states, by allowing nodes only to enter $\mathcal{S}$ appropriately (so that inconsistencies are not introduced), to reach the legitimate state to obtain the maximal 2-packing. The algorithm assumes a synchronous scheduler where at any step all privileged nodes are selected to move. The underlying approach consists of two logical sets of actions:

4.3.1.1 Exit $\mathcal{S}$

A node $i \in \mathcal{S}$ exits $\mathcal{S}$ in a step of execution of the algorithm iff at least one neighbor $j \in N(i)$ is deemed to be inconsistent (Remark 4.3.1.2), as evidenced by the content of the variable c_j; the rationale is that exiting of i may not decrease but never increase the number of inconsistent nodes $j \in N[i]$ even c_j's are erroneous in a system state causing i to exit $\mathcal{S}$ (Remark 4.3.1.2). Also, by the same reason, simultaneous exit of multiple nodes from $\mathcal{S}$ cannot increase the number of inconsistent nodes in the system; our objective is to reach a safe state (Definition 4.3.1.2) as quickly as possible starting from an illegitimate state.

Definition 4.3.2 *For a node i, a Boolean predicate* $\texttt{nowExit}_i = 1$ *iff $i \in \mathcal{S}$ and at least one of its neighbors is deemed to be inconsistent (Remark 4.3.1.2):*

$$\texttt{nowExit}_i \stackrel{\text{def}}{\equiv} (s_i = 1) \wedge (\exists j \in N(i) : c_j \geq 2)$$

4.3.1.2 Enter $\mathcal{S}$

The algorithm requires that *after a node $i \notin \mathcal{S}$ enters $\mathcal{S}$ in a step, node i is guaranteed to be the unique $\mathcal{S}$ node within its 2-hop neighborhood at the end of current step*, and thus each node $j \in N[i]$ remains consistent after node i enters $\mathcal{S}$. To accomplish this requirement we use a locking mechanism and delay technique.

A locking mechanism (implemented by stored variable p_i) is employed such that when a node enters $\mathcal{S}$ in a step, all other nodes in its 2-hop neighborhood are prohibited to enter $\mathcal{S}$. Specifically, when a node i needs to enter $\mathcal{S}$, it first requests a lock by setting a self-pointer (i.e., $p_i = i$). A node i is *locked* or gets the lock iff all nodes in $N[i]$ point to i. The neighbor j of i grants the lock by updating its pointer to i, i.e., $p_j = i$. It is possible that two adjacent nodes request locks simultaneously. In order to break the tie, the node grants the lock (by updating its pointer) to the smallest ID neighbor with self-pointer.

After a node i becomes locked, it sets d bit to delay its Enter move by one step only. It should be noted that in delayed step: (a) each node $j \in N[i]$ must have no locked neighbor except i (See Definition 4.3.3.2 below) and hence no neighbor entering $\mathcal{S}$; (b) each neighbor $j \in N[i]$ gets chance to update c_j, such that $c_j \geq |N[j] \cap \mathcal{S}|$ are guaranteed when node i enters (it is possible that some neighbor of $j \in N[i]$ exits $\mathcal{S}$ in delayed step, hence the inequality). After the delayed step, node i is guaranteed to be safe to enter (i.e., after node i enters $\mathcal{S}$, it is the unique $\mathcal{S}$ node within its 2-hop neighborhood of node i, and hence each node $j \in N[i]$ remains consistent).

In the following two definitions, a few predicates are defined to facilitate the stepwise development of the proposed algorithm.

Definition 4.3.3 *In a system state, a node i can locally compute each of the following Boolean predicates:*

1. *For a node i, a Boolean predicate* $\texttt{needEnter}_i = 1$ *iff $i \notin \mathcal{S}$ and there does not exist $\mathcal{S}$ node within distance-2 of node i, as evidenced by the content of the variable c_j on each $j \in N(i)$:*

$$\texttt{needEnter}_i \stackrel{\text{def}}{\equiv} (s_i = 0) \wedge (\forall j \in N(i) : c_j = 0)$$

2. *To implement* ***locking mechanism****, two more Boolean predicates* $\texttt{requestLock}_i$ *and* $\texttt{locked}_i$

on node i are defined as:

$$\texttt{requestLock}_i \stackrel{\text{def}}{\equiv} \texttt{needEnter}_i \wedge (\forall j \in N[i] : p_j = null)$$

$$\texttt{locked}_i \stackrel{\text{def}}{\equiv} \forall j \in N[i] : p_j = i$$

Note: *In a system state, if node i is locked, no node in $N^2(i)$ can be locked in the same state.*

3. *In any system state, a node i* ***requests a delay*** *iff the Boolean predicate $\texttt{requestDelay}_i = 1$ where*

$$\texttt{requestDelay}_i \stackrel{\text{def}}{\equiv} (d_i = 0) \wedge \texttt{needEnter}_i \wedge \texttt{locked}_i$$

4. *In any system state, a node i can* ***enter*** $\mathcal{S}$ *iff the Boolean predicate $\texttt{nowEnter}_i = 1$ where*

$$\texttt{nowEnter}_i \stackrel{\text{def}}{\equiv} (d_i = 1) \wedge \texttt{needEnter}_i \wedge \texttt{locked}_i$$

Note: *In any system state, if a node i is ready to enter ($\texttt{nowEnter}_i = 1$), no node in $N^2(i)$ is ready to enter in the same state since no node in $N^2(i)$ can be locked in that state (Definition 4.3.3.2).*

Definition 4.3.4 *For a node i in any system state:*

1. *The predicate $\texttt{updateC}_i$ is true iff its c_i is not correct, i.e.,*

$$\texttt{updateC}_i \stackrel{\text{def}}{\equiv} c_i \neq |N[i] \cap \mathcal{S}|$$

2. *The predicate $\texttt{updateP}_i$ is true iff its p_i is not equal to $minSP_i$, i.e.,*

$$\texttt{updateP}_i \stackrel{\text{def}}{\equiv} p_i \neq minSP_i$$

3. *The predicate $\texttt{clearD}_i$ is true iff $d_i = 1$ and it does not need to enter $\mathcal{S}$ or is not locked, i.e.,*

$$\texttt{clearD}_i \stackrel{\text{def}}{\equiv} (d_i = 1) \wedge \neg(\texttt{needEnter}_i \wedge \texttt{locked}_i)$$

4. *The predicate* $\texttt{releaseLock}_i$ *is true iff it has the self pointer and its pointer is equal to* $minSP_i$ *but does not need to enter* $\mathcal{S}$*, i.e.,*

$$\texttt{releaseLock}_i \stackrel{\text{def}}{\equiv} \neg\texttt{needEnter}_i \wedge (p_i = i) \wedge (minSP_i = i)$$

The complete pseudo code of algorithm M2PSC is shown in Figure 4.2. A few simple characteristics of the algorithm are highlighted in the following remark.

```
RA: if nowExit_i ∨ requestLock_i ∨ releaseLock_i ∨ updateP_i ∨ updateC_i ∨ clearD_i
      then { if nowExit_i
               then s_i ← 0;                                   [Exit S]
             if requestLock_i
               then p_i ← i;                                   [Request_Lock]
             if releaseLock_i
               then p_i ← null;                                [Release_Lock]
             if updateP_i
               then p_i ← minSP_i;                             [Update_Pointer]
             c_i ← |N[i] ∩ S|;                                 [Update_Counter]
             d_i ← 0;                                          [Clear_Delay]
RB: if requestDelay_i
      then { d_i ← 1; c_i ← |N[i] ∩ S|;                        [Request_Delay]
RC: if nowEnter_i
      then { s_i ← 1; p_i ← null; d_i ← 0; c_i ← |N[i] ∩ S|;   [Enter S]
```

Figure 4.2: Algorithm M2PSC on node i, $1 \leq i \leq n$

Remark 4.3.2 *In a given step* r*,* $r \geq 1$*, of execution:*

1. *If node* i *has incorrect* c_i *in* Σ_{r-1}*, it must update* c_i *in step* r*.*

2. *For a node* i*,* $\texttt{nowEnter}_i$*,* $\texttt{requestDelay}_i$ *and* $\texttt{nowExit}_i$ *are pairwise mutual exclusive;* $\texttt{requestLock}_i$*,* $\texttt{releaseLock}_i$ *and* $\texttt{updateP}_i$ *are pairwise mutual exclusive.*

3. *The membership of node* i *is changed only by rules* RA *(Exit* $\mathcal{S}$ *move) and* RC *(Enter* $\mathcal{S}$ *move). If a node* i *is privileged to make Exit* $\mathcal{S}$ *move, it must exit* $\mathcal{S}$ *successfully under synchronous scheduler (see part(2)).*

4. *If node* i *exits* $\mathcal{S}$*, its neighboring nodes can concurrently exits* $\mathcal{S}$ *if they are eligible to do so; If node* i *enters* $\mathcal{S}$*, no node* $j \in N^2(i)$ *can concurrently enter* $\mathcal{S}$ *in the same step (Definition 4.3.3.4).*

5. *A node i can acquire a self-pointer ($p_i = i$) only by making Request_Lock move in rule* RA *when it needs to enter $\mathcal{S}$ to maximize $|\mathcal{S}|$ and all its neighbors have null pointers.* ***Note****: a node cannot acquire a self-pointer by making Update_Pointer move in rule* RA.

6. *A node i releases its self-pointer when it does not need to enter $\mathcal{S}$ ($\texttt{needEnter}_i = 0$) by making Release_Lock move, or when it has at least one smaller ID neighbor with self-pointer by making Update_Pointer move.*

7. *After a node i with $\texttt{needEnter}_i = 1$ becomes locked, i.e., $\texttt{locked}_i = 1$, it delays its enter move by one step only by making Request_Delay move (setting $d_i = 1$) such that its neighbors have time to correct their c-variables.*

8. *If $d_i = 1$ in Σ_{r-1}, then node i will clear delay by making either Enter $\mathcal{S}$ or Clear_Delay move such that $d_i = 0$ in Σ_r (Definitions 4.3.3.4 and 4.3.4.3).*

Definition 4.3.5 *In any system state $\Sigma_r, r \geq 0$:*

1. *A node i is* ***privileged*** *if it is enabled by any of the rules of the algorithm.*

2. *The execution of the algorithm* ***terminates*** *when no node is privileged.*

We first prove that $\mathcal{S}$ is a maximal 2-packing when algorithm M2PSC terminates, and then we show the algorithm is safely converging in the sense that starting from an arbitrary state, it first converges to a 2-packing (a safe state) in 3 steps, and then stabilizes in a maximal one (the legitimate state) in $O(n)$ steps without breaking safety, where n is the number of nodes.

Lemma 4.3.1 *If algorithm M2PSC terminates, then for each node $i \in V$*

(a) c_i is correct, i.e., $c_i = |N[i] \cap \mathcal{S}|$.

(b) $d_i = 0$.

(c) $p_i = null$.

(d) $\texttt{nowExit}_i = 0$ and $\texttt{needEnter}_i = 0$.

Proof: (a) This lemma immediately follows from the fact that no node is privileged by the rule RA at the termination of the algorithm.

(b) Assume, by contradiction, there exists some node(s) j with $d_j = 1$. Node j must have $\texttt{needEnter}_j = 1$ and $\texttt{locked}_j = 1$ (otherwise node j is privileged by rule RA to make Clear_Delay move). Thus, node j is privileged by rule RC, a contradiction.

(c) If no node j has self-pointer, then $minSP_i = null$ for all $i \in V$ and hence the lemma holds since $p_i = minSP_i$ (otherwise node i is privileged by rule RA to make Update_Pointer move). So the key point here is to show that there is no node with self-pointer. Assume, by contradiction, there exists some node(s) with self-pointer. Consider the node with minimum ID from among those nodes, say node j; $minSP_k = j$ for each node $k \in N[j]$. Also each node $k \in N[j]$ must have $p_k = minSP_k = j$ (otherwise node k would be privileged by the rule RA to make Update_Pointer move). Thus, node j is locked (i.e., $\texttt{locked}_j = 1$). Also, node j must have $\texttt{needEnter}_j = 1$ (otherwise node j is privileged by rule RA to make Release_Lock move). Thus, we get node j is privileged by rule RB to make Request_Delay move (by part(b)), a contradiction.

(d) No node i is privileged by rule RA to make Exit $\mathcal{S}$ move and Request_lock move; the claim follows from parts (a) and (c). □

Theorem 4.3.1 *Starting from an arbitrary system state, if algorithm M2PSC terminates using synchronous scheduler, then $\mathcal{S}$ is a maximal 2-packing.*

Proof: First, we show $\mathcal{S}$ is a 2-packing. Assume, by contradiction, $\mathcal{S}$ is not 2-packing, i.e., there exists some node(s) i such that $|N[i] \cap \mathcal{S}| \geq 2$, i.e., $c_i \geq 2$. Thus $\texttt{nowExit}_j = 1$ for each $j \in N(i) \cap \mathcal{S}$; node j is privileged by the rule RA (Exit $\mathcal{S}$), a contradiction. Thus $\mathcal{S}$ is a 2-packing.

Next, we claim $\mathcal{S}$ is maximal. Assume otherwise, i.e., there exist some node(s) $i \in \{V - \mathcal{S}\}$ such that $\nexists j \in \mathcal{S} : dist(i, j) \leq 2$. Thus $\texttt{needEnter}_i = 1$ and node i is privileged by RA to make Request_Lock move (by Lemma 4.3.1(c)), a contradiction. □

Lemma 4.3.2 *In any system state Σ_r, $r \geq 1$, if a node i is enabled by the rule RC to enter $\mathcal{S}$, each node $j \in N[i]$ must have $c_j \geq |N[j] \cap \mathcal{S}|$.*

Proof: In the system state Σ_{r-1}, node i must have had $d_i = 0$, $\texttt{needExit}_i = 1$ and $\texttt{locked}_i = 1$; otherwise it is impossible for node i to have $d_i = 1$ in Σ_r (Remark 4.3.2.8 and Request_Delay move). Since $\texttt{locked}_i = 1$ in Σ_{r-1}, no node $j \in N^2(i)$ was locked (Definition 4.3.3.2) and hence entered $\mathcal{S}$ in step r. Coupled with the fact that each node $j \in N[i]$ corrected its c_j in step r (Remark 4.3.2.1), thus, in Σ_r, each node $j \in N[i]$ must have $c_j \geq |N[j] \cap \mathcal{S}|$. **Note:** it is possible that some neighbor of node j exits $\mathcal{S}$ in step r (hence the inequality). □

Lemma 4.3.3 *In step $r \geq 2$, if a node i enters $\mathcal{S}$ (by executing rule* RC*), each node $j \in N[i]$ has $|N[j] \cap \mathcal{S}| = 1$ at the end of step.*

Proof: If node i enters $\mathcal{S}$ in step r, then node $j \in N[i]$ must have had $c_j = 0$ in Σ_{r-1} (Definition 4.3.3.4), and thus $|N[j] \cap \mathcal{S}| = 0$ by Lemma 4.3.2. Coupled with the fact that no other neighbors of node $j \in N[i]$ can enter $\mathcal{S}$ in the same step by Definition 4.3.3.4, the lemma holds. □

Lemma 4.3.4 *At the end of step $r \geq 2$, if there exists some node(s) i such that $|N[i] \cap \mathcal{S}| \geq 2$, then c_i must be ≥ 2.*

Proof: Consider a node i with $|N[i] \cap \mathcal{S}| \geq 2$ at the end of step $r \geq 2$. We observe that no neighbor of node i entered $\mathcal{S}$ in step r (otherwise $|N[i] \cap \mathcal{S}|$ would be 1 at the end of step r by Lemma 4.3.3). But some neighbor(s) of node i may exit $\mathcal{S}$ in step r. Thus node i must have $c_i \geq |N[i] \cap \mathcal{S}| \geq 2$ at the end of step r by Remark 4.3.2.1. □

Theorem 4.3.2 *Starting from any initial illegitimate state, algorithm* M2PSC *converges to a safe state ($\mathcal{S}$ denotes a 2-packing, i.e., each node i has $|N[i] \cap \mathcal{S}| \leq 1$) after 3 steps.*

Proof: We show that each node i has $|N[i] \cap \mathcal{S}| \leq 1$ after 3 steps of execution; we consider three cases:

(a) Consider any node i with $|N[i] \cap \mathcal{S}| = 0$ in Σ_2: If any node $j \in N[i]$ enters $\mathcal{S}$ in the step 3, by Lemma 4.3.3 node i will still have $|N[i] \cap \mathcal{S}| \leq 1$ after step 3.

(b) Consider any node i with $|N[i] \cap \mathcal{S}| = 1$ in Σ_2: no node $j \in N[i]$ enters $\mathcal{S}$ in the step 3 (Definition 4.3.3.4 and Lemma 4.3.2), node i will still have $|N[i] \cap \mathcal{S}| \leq 1$ after step 3.

(c) Consider any node i with $|N[i] \cap \mathcal{S}| \geq 2$ in Σ_2: c_i must be ≥ 2 by Lemma 4.3.4, each neighbor j of node i has $\texttt{nowExit}_j = 1$. Thus, in the step 3 each neighbor j of node i must exit $\mathcal{S}$ by executing the rule RA (Remark 4.3.2.3). It follows that node i will have $|N[i] \cap \mathcal{S}| \leq 1$ after step 3. □

Theorem 4.3.3 *After step 3, algorithm* M2PSC *maintains safety in all subsequent steps before converging to a legitimate state.*

Proof: After step 3, the system reaches a safe state. In any safe state, any node i in V has $|N[i] \cap \mathcal{S}| \leq 1$. If any neighbor of i enters $\mathcal{S}$, node i will remain having $|N[i] \cap \mathcal{S}| \leq 1$ in the next state by Lemma 4.3.3, thus the system reaches another safe state. □

Lemma 4.3.5 *Starting from a safe state Σ_r, $r \geq 4$, no node will ever make Exit $\mathcal{S}$ move in subsequent steps.*

Proof: In a safe state, each node $i \in V$ has $|N[i] \cap \mathcal{S}| \leq 1$. The algorithm `M2PSC` always in the safe state after step 3 (Theorem 4.3.3), thus each node i always has $|N[i] \cap \mathcal{S}| \leq 1$ after step 3 and each node i always has $c_i \leq 1$ after step 4 by Remark 4.3.2.1. The lemma holds. □

Lemma 4.3.6 *In any safe state Σ_r, $r \geq 5$, rules* RA, RB *and* RC *are pairwise mutual exclusive.*

Proof: It is easy to show `requestDelay` and `nowEnter` are pairwise mutual exclusive with each of `nowExit`, `requestLock`, `releaseLock`, `updateP` and `clearD`, we here omit the details. Coupled with the fact that `requestDelay` and `nowEnter` are pairwise mutual exclusive (Remark 4.3.2.2). Thus, to prove the lemma, it suffices that show that `requestDelay` and `nowEnter` are pairwise mutual exclusive with `updateC`.

For any node i, if $\texttt{requestDelay}_i = 1$ or $\texttt{nowEnter}_i = 1$ in Σ_r, no node $j \in N[i]$ entered $\mathcal{S}$ in the step r (otherwise node i cannot have self-pointer in Σ_r). Coupled with Lemma 4.3.5 and Remark 4.3.2.1, c_i must be correct in Σ_r. Thus, $\texttt{requestDelay}_i$ and $\texttt{nowEnter}_i$ are pairwise mutual exclusive with $\texttt{updateC}_i$ in Σ_r. □

Lemma 4.3.7 *In any system state Σ_r where $r \geq 1$, two adjacent nodes i and j have self-pointers (i.e., $p_i = i$ and $p_j = j$), then (a) the larger ID node will lose the self-pointer (i.e., if say $i < j$, $p_j \neq j$) in the next steps; (b) nodes i and j must have concurrently acquired the self-pointers in step r.*

Proof: (a) In Σ_r, node j has $p_j = j \neq minSP_j$ and thus $\texttt{UpdateP}_j = 1$. Coupled with the fact that $\texttt{UpdateP}_j$ is pairwise mutual exclusive with each of $\texttt{requestDelay}_j$ and $\texttt{nowEnter}_j$, in the next step node j must be selected by scheduler to make Update_Pointer move in rule RA. (b) If $p_i = i$ but $p_j \neq j$, then node j cannot make Request_Lock move to get the self-pointer in the next state (Definition 4.3.3.2). □

Lemma 4.3.8 *In a safe state Σ_r, $r \geq 3$, for two adjacent nodes i and j, if $p_i = i$ and $p_j = j$, then* $\texttt{needEnter}_i = \texttt{needEnter}_j = 1$.

Proof: It follows from Lemma 4.3.7 that nodes i and j had concurrently made Request_Lock move to get self-pointers in step r. In Σ_{r-1}, $\texttt{needEnter}_i = 1$, $\texttt{needEnter}_j = 1$, $p_i = p_j = null$, and

for each $k \in N(i) \cup N(j)$, $p_k = null$ (nodes i and j are enabled for rule RA to make Request_Lock move; Definition 4.3.3.2). We argue that:

(a) Node i can not enter $\mathcal{S}$ (execute rule RC) in step r since $\texttt{locked}_i = 0$ in Σ_{r-1} ($p_i = null$).

(b) Any node $k \in N(i)$ cannot enter $\mathcal{S}$ (execute rule RC) in step r since $\texttt{locked}_k = 0$ in Σ_{r-1} ($p_k = null$).

(c) Any neighbor k' of $k \in N(i)$ cannot enter $\mathcal{S}$ (execute rule RC) in step r since $\texttt{locked}_{k'} = 0$ in Σ_{r-1} ($p_k = null$).

Thus, $\texttt{needEnter}_i$ remains 1 in Σ_r (by similar reasoning, $\texttt{needEnter}_j$ remains 1 in Σ_r). □

Definition 4.3.6 *In any system state,*

1. *We define an island $\mathcal{I}$ to be a* maximal *set of nodes $\{i \in V | \texttt{needEnter}_i = 1 \wedge p_i = i\}$ such that the subgraph of G induced by the set $\mathcal{I}$ is connected.*

2. *We use $\boldsymbol{\alpha}$ to denote the number of islands and $\boldsymbol{\beta}$ to denote the number of nodes i with $\texttt{needEnter}_i = 1$.*

Remark 4.3.3 *In any system state:*

1. *An island may consist of a single or multiple nodes; a node i with $\texttt{needEnter}_i = 1$ and $p_i \neq i$ is not a member of any island.*

2. *For a node i in an island of size ≥ 2, $\texttt{locked}_i = 0$ since it has a neighbor j with $p_j = j \neq i$ (Definition 4.3.3.2).*

3. *$\alpha \leq \beta$; $\alpha < n$; $\beta \leq n$;*

4. *After step 4, β is non increasing in subsequent steps (Definition 4.3.3.1 and Lemma 4.3.5).*

5. *When algorithm M2PSC terminates, $\alpha = \beta = 0$.*

Lemma 4.3.9 *If a node i enters $\mathcal{S}$ (by executing rule* RC*) in a step, node i constitutes a single node island at the beginning of the step.*

Proof: Node i enters $\mathcal{S}$; thus $\texttt{needEnter}_i = 1$ and $\texttt{locked}_i = 1$ (rule RC). Since $p_i = i$, node i belongs to an island (Definition 4.3.6.1); node i does not belong to an island of size ≥ 2 (Remark 4.3.3.2). □

Lemma 4.3.10 *In any step r, $r \geq 5$ (starting from a safe state Σ_{r-1}), (a) α can not decrease if β remains constant; (b) α decreases at least by 1 and at most by ℓ, if β decreases by ℓ ($1 \leq \ell \leq \beta$).*

Proof: (a) If β remains constant, no node i changes $\texttt{needEnter}_i$ from 1 to 0. (1) Any island $\mathcal{I}$ cannot disappear since the smallest ID node in $\mathcal{I}$, say node i, cannot change its pointer in step r ($p_i = i = minSP_i$ in Σ_{r-1} [no neighbor j of i with $\texttt{needEnter}_j = 0$ has a self-pointer by Lemma 4.3.8 and node i does not have any island node neighbor with a smaller ID]). (2) Two islands cannot merge into one: consider any 2 islands $\mathcal{I}_1$ and $\mathcal{I}_2$; since $\mathcal{I}_1 \cup \mathcal{I}_2 = \emptyset$, for the two islands to merge, there must be a node $j \in N(\mathcal{I}_1 \cup \mathcal{I}_2)$ such that $\texttt{needEnter}_j = 1$ in Σ_{r-1} and node j acquires self-pointer in Σ_r (j becomes an island node in Σ_r) by executing rule RA to make Request_Lock move in step r; this is impossible since j has neighbor(s) with non null pointers in Σ_{r-1} (see Definition 4.3.3.2).

(b) Starting in a safe state Σ_{r-1}, if β decreases by ℓ in Σ_r, ℓ nodes have changed their needEnter bits from 1 to 0. Consider any node i whose $\texttt{needEnter}_i$ is changed from 1 to 0. At least one of the two must occur in step r (Definition 4.3.3.1): (1) node i enters $\mathcal{S}$ by executing rule RC; (2) some neighbor(s) of node $j \in N(i)$ enters $\mathcal{S}$ by executing rule RC such that $c_j > 0$. If all ℓ nodes change their needEnter from 1 to 0 because of (1), then α is decreased by ℓ by Lemma 4.3.9; If some node(s) i changes $\texttt{needEnter}_i$ from 1 to 0 because of (2), then it is possible that node i does not belong to any island. Although the change of $\texttt{needEnter}_i$ on node i causes β to decrease in Σ_r, it does not cause α to decrease (if node i is not an island node in Σ_{r-1}); but, for the possibilities (2), at least some other node must enter $\mathcal{S}$ (change s bit to 1) by executing rule RC in the step, thereby causing α to decrease (Lemma 4.3.9). Thus, α decreases by at most ℓ and at least by 1 in Σ_r. □

Lemma 4.3.11 *Starting from a safe state Σ_r, $r \geq 4$, with $\beta \neq 0$,*

(a) either α increases in at most 3 next steps, if β remains constant;

(b) or β decreases in at most 4 next steps.

Proof: Starting from a safe state Σ_r, $r \geq 4$, any node i with $\texttt{needEnter}_i = 0$ and $p_i = i$ must either make Update_Pointer or Release_Lock move in step $r+1$ to make $p_i \neq i$ in Σ_{r+1}. Then, Σ_{r+1} does not have a node j with $\texttt{needEnter}_j = 0 \wedge p_j = j$ (otherwise, node j had $\texttt{needEnter}_j = 1$ in Σ_r and hence β has decreased by at least 1 in one step). There are two possibilities:

(1) **There is no island node:** In Σ_{r+1}, each node i has $minSP_i = null$ (no node with self-pointer and Definition 4.3.1.3). Also, since $\beta \neq 0$, there must be a node k with $\texttt{needEnter}_k = 1$; node k, in the worst case, must make Update_Pointer move in step $r+2$ and Request_Lock move in step $r+3$ in that sequence to get $p_k = k$. Thus, there is a new island $\{k\}$, i.e., α has increased in at most 3 steps starting in Σ_r.

(2) **There is at least one island node**: If there are multiple such island nodes, let i be the node with minimum ID among those. In the worst case, each node $j \in N(i)$ makes Update_Pointer move in step $r+2$ to update their pointers to i; node i becomes locked, i.e., $\texttt{locked}_i = 1$ (Definition 4.3.3.2), in Σ_{r+2}. Also, if $d_i = 1$ in Σ_{r+1}, node i makes Clear_Delay move in step $r+2$ such that $d_i = 0$ in Σ_{r+2}. Now, there are two possibilities:

(i) At least one $j \in N(i)$ has $minSP_j = k$ in Σ_{r+2} where $k \in N(j)$ (Definition 4.3.1.3), $p_k = k$, and $k < i$. Node k must have acquired its self-pointer by making Request_Lock move in step $r+2$ and so, $\texttt{needEnter}_k = 1 \wedge (\forall k' \in N(k) : p_{k'} = null)$ in Σ_{r+1}, i.e., node k is not connected to any island nodes. Thus, $\{k\}$ is a newly formed single node island in Σ_{r+2}, i.e., α increases in at most 2 steps.

(ii) Each $j \in N(i)$ has $minSP_j = i$ in Σ_{r+2}; in step $r+3$, node i makes Request_Delay move to delay its Enter move by one step only. Thus, node i executes rule RC to enter $\mathcal{S}$ in step $r+4$; so $\texttt{needEnter}_i = 0$ in Σ_{r+4}, i.e., β decreases in at most 4 steps.

□

Lemma 4.3.12 *After step 4, algorithm M2PSC reaches a safe state with $\alpha = \beta = 0$ in at most $7n$ steps under a synchronous scheduler.*

Proof: In a safe state where $\alpha = \beta$, if β decreases by 1, α must decrease by 1 (Lemma 4.3.10); $\beta \leq n$ and β is non-increasing (Remarks 4.3.3.3 and 4.3.3.4). Recall that β decreases by at least 1 in at most 4 steps (Lemma 4.3.11(b)). Thus, from any safe system state with $\alpha = \beta$, the system will be in a safe state with $\alpha = \beta = 0$ in at most $4n$ steps. Also, if β remains constant, α must increase by 1 in at most 3 steps (Lemma 4.3.11(a)); in at most $3n$ steps, α will be equal to β. Thus, the system will be in a safe state with $\alpha = \beta = 0$ in at most $4n + 3n = 7n$ steps. □

Theorem 4.3.4 *Starting in any arbitrary state, the algorithm M2PSC terminates in at most $O(n)$ steps under a synchronous scheduler.*

Proof: The system reaches a safe state with $\alpha = \beta = 0$ in $7n + 4 = O(n)$ steps in the worst case (Theorem 4.3.2 and Lemma 4.3.12). In the next step: all node pointers will be *null*, all c variables will be correct and all d variables will be 0; thus the algorithm terminates. □

4.3.2 Maximal k-packing with Safe Convergence

The basic idea of algorithm `M2PSC` is generalized to obtain a self-stabilizing maximal k-packing algorithm with safe convergence (called algorithm `MKPSC`). As before, starting from an arbitrary state, the system first converges to a k-packing (a safe state) by allowing nodes to exit $\mathcal{S}$ quickly, and thereafter moves through safe states until $\mathcal{S}$ is a maximal k-packing (the legitimate state) by allowing nodes only to enter $\mathcal{S}$ appropriately. The algorithm assumes a synchronous scheduler where in any step all privileged nodes are selected to move. In algorithm `MKPSC`, each node $i, 1 \leq i \leq n$, maintains the following variables:

- An array of nonnegative integer sets $\hat{\boldsymbol{T}}_{\boldsymbol{i}}\boldsymbol{[0, ..., k-1]}$; $\hat{T}_i[\ell]$, $0 \leq \ell < k$, is intended to keep track of the IDs of $\mathcal{S}$-nodes (as a set) in $N^{\ell}[i]$, ℓ-hop closed neighborhood of node i in any system state. In a system state $\mathcal{S}$ is the current set of nodes i with $\hat{T}_i[0] = \{i\}$.

- A pointer array $\hat{\boldsymbol{P}}_{\boldsymbol{i}}\boldsymbol{[0, ..., k-1]}$; $\hat{P}_i[\ell]$ (which may be null) points to a node j, indicated by $\hat{P}_i[\ell] = j$. We say node i has a **self-pointer** iff $\hat{P}_i[0] = i$; $\hat{P}_i[\ell]$, $0 \leq \ell < k$, keeps track of the minimum ID node with self-pointer in $N^{\ell}[i]$ at any system state.

- A nonnegative integer variable $\boldsymbol{d_i}$; node i uses this variable to delay some activity by $2k$ steps.

Remark 4.3.4 *In any system state, for any node i, (a) $\hat{T}_i$ is **correct** iff (1) $\hat{T}_i[0]$ is either $\{i\}$ or $\emptyset$, and (2) for each $\ell, 1 \leq \ell \leq (k-1)$, $\hat{T}_i[\ell] = \underset{j \in N[i]}{\cup} \hat{T}_j[\ell - 1]$; (b) $\hat{P}_i$ is **correct** iff (1) $\hat{P}_i[0]$ is either i or null, and (2) for each $\ell, 1 \leq \ell \leq (k-1)$, $\hat{P}_i[\ell] = \min_{j \in N[i]} \hat{P}_j[\ell - 1]$.*

Definition 4.3.7 *A system state is **safe** if $\mathcal{S} = \{i | i \in V \wedge \hat{T}_i[0] = \{i\}\}$ denotes a k-packing, i.e, each node $i \in \mathcal{S}$ is the unique $\mathcal{S}$-node within distance-k of node i (Definition 1.3.7). A system state is **legitimate** if $\mathcal{S}$ denotes a maximal k-packing.*

The underlying approach consists of two logical sets of actions: (a) **Exit $\boldsymbol{\mathcal{S}}$**: The algorithm requires that A node $i \in \mathcal{S}$ exits $\mathcal{S}$ in a step of execution of the algorithm iff there exists some $\mathcal{S}$-node(s) within distance-k of node i, as evidenced by the content of the variable $\hat{T}_j[k-1]$ at each

node $j \in N(i)$. This quick exit of nodes from $\mathcal{S}$ facilitates quick convergence to a safe state. (b) **Enter $\mathcal{S}$**: Once in a safe state, a node i needs to enter $\mathcal{S}$ without violating safety in the process to eventually converge to a legitimate state ($\mathcal{S}$ is a maximal k-packing). The algorithm requires that *after a node $i \notin \mathcal{S}$ enters $\mathcal{S}$ in a step, node i is guaranteed to be the unique $\mathcal{S}$-node within distance-k of node i at the end of current step.* To accomplish this requirement we generalize the concepts of locking mechanism and delay technique in algorithm `M2PSC` such that (1) when a node enters $\mathcal{S}$ in a step, all other nodes in its k-hop neighborhood are prohibited to enter $\mathcal{S}$ in the same step; (2) no $\mathcal{S}$-node exists in k-hop neighborhood of node i at the beginning of the step, i.e., after node i intends to enter $\mathcal{S}$, it must communicate with nodes in its k-hop neighborhood to make sure $|N^{k-1}[j] \cap \mathcal{S}| = 0$ for each $j \in N[i]$ and none of those nodes enter $\mathcal{S}$; (3) in case more than one node in a k-hop neighborhood intends to enter $\mathcal{S}$, a tie resolution mechanism is needed. The entire process takes $2k$ steps in the proposed algorithm and is facilitated by the delay variable d_i. We need to define a few predicates to facilitate the stepwise development of the proposed algorithm.

Definition 4.3.8 *In any system state, a Boolean predicate* $\texttt{nowExit}_i = 1$ *for a node i is defined as*

$$\texttt{nowExit}_i \stackrel{\text{def}}{\equiv} (\hat{T}_i[0] = \{i\}) \wedge (\exists j \in N(i) : |\hat{T}_j[k-1]| \geq 2)$$

Definition 4.3.9 *In a system state, for a node i*

1. *The Boolean predicate* $\texttt{needEnter}_i = 1$ *iff $i \notin \mathcal{S}$ and there does not exist $\mathcal{S}$-node within distance-k of node i, as evidenced by the content of the variable $\hat{T}_j[k-1]$ on each $j \in N(i)$:*

$$\texttt{needEnter}_i \stackrel{\text{def}}{\equiv} \big(\hat{T}_i[0] \neq \{i\}\big) \wedge \big(\forall j \in N(i) : |\hat{T}_j[k-1]| = 0\big)$$

2. *A node i requests a lock (to express its intent to enter $\mathcal{S}$) when its own $\hat{P}_i[0]$ is null as well as all nodes in $N[i]$ have their respective pointers (for distance $k-1$) are null. A node i is locked when it has a self-pointer as well as all nodes in $N[i]$ have their $k-1$-distance pointers pointing to i.*

$$\texttt{requestLock}_i \stackrel{\text{def}}{\equiv} \texttt{needEnter}_i \wedge (\hat{P}_i[0] = \mathit{null}) \wedge (\forall j \in N[i] : \hat{P}_j[k-1] = \mathit{null})$$

$$\texttt{locked}_i \stackrel{\text{def}}{\equiv} (\hat{P}_i[0] = i) \wedge (\forall j \in N[i] : \hat{P}_j[k-1] = i)$$

3. *A node i needs to* ***request a delay*** *when it needs to enter $\mathcal{S}$ and is locked and has not yet waited for $2k$ steps to ensure correctness of $\hat{T}_i$ variables, i.e.,*

$$\texttt{requestDelay}_i \stackrel{\text{def}}{\equiv} (d_i \neq 2k) \wedge \texttt{needEnter}_i \wedge \texttt{locked}_i$$

4. *A node i can immediately* ***enter $\mathcal{S}$*** *when it has waited for $2k$ steps maintaining its readiness to enter and locked, i.e., iff the Boolean predicate $\texttt{nowEnter}_i = 1$ where*

$$\texttt{nowEnter}_i \stackrel{\text{def}}{\equiv} (d_i = 2k) \wedge \texttt{needEnter}_i \wedge \texttt{locked}_i$$

Definition 4.3.10 *For a node i in any system state:*

1. *The predicate $\texttt{update}\hat{T}_i$ is true iff its $\hat{T}_i$ is not correct (Remark 4.3.4), i.e.,*

$$\texttt{update}\hat{T}_i \stackrel{\text{def}}{\equiv} \left(\hat{T}_i[0] \neq \{i\} \wedge \hat{T}_i[0] \neq \emptyset\right) \vee \left(\hat{T}_i[\ell] \neq \bigcup_{j \in N[i]} (\hat{T}_j[\ell - 1]) \text{ for some } \ell \in \{1, 2, ..., k-1\}\right)$$

2. *The predicate $\texttt{update}\hat{P}_i$ is true iff its $\hat{P}_i$ is not correct (Remark 4.3.4), i.e.,*

$$\texttt{update}\hat{P}_i \stackrel{\text{def}}{\equiv} (\hat{P}_i[0] \neq i \wedge \hat{P}_i[0] \neq null) \vee \left(\hat{P}_i[\ell] \neq \min_{j \in N[i]} (\hat{P}_j[\ell - 1]) \text{ for some } \ell \in \{1, 2, ..., k-1\}\right)$$

3. *The predicate $\texttt{clearD}_i$ is true if the delay indicator $d_i \neq 0$ and either the node i is not eligible to enter $\mathcal{S}$ or it is not locked, i.e.,*

$$\texttt{clearD}_i \stackrel{\text{def}}{\equiv} (d_i \neq 0) \wedge \neg(\texttt{needEnter}_i \wedge \texttt{locked}_i)$$

4. *The predicate $\texttt{releaseLock}_i$ is true iff it has the self pointer but it is not eligible to enter $\mathcal{S}$, i.e.,*

$$\texttt{releaseLock}_i \stackrel{\text{def}}{\equiv} \neg\texttt{needEnter}_i \wedge (\hat{P}_i[0] = i)$$

The complete pseudo code of algorithm MKPSC is shown in Figure 4.3. A few simple characteristics of the algorithm are highlighted in the following remark.

Remark 4.3.5 *In a given step r, $r \geq 1$, of execution:*

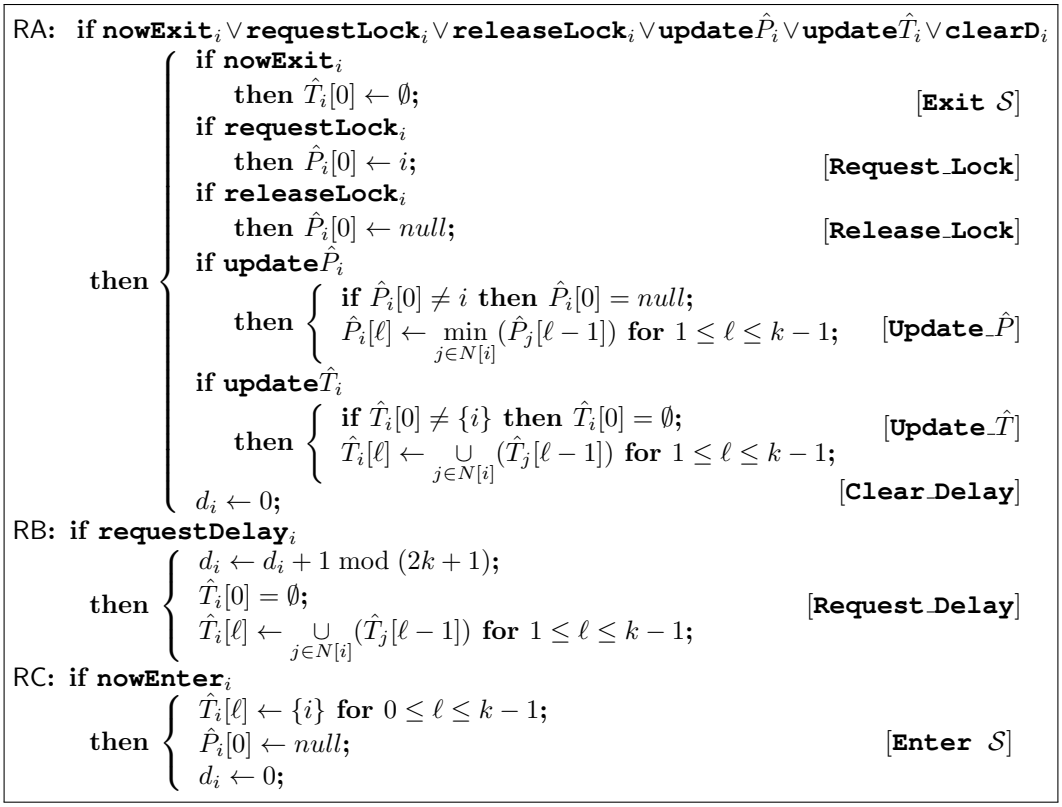

Figure 4.3: Algorithm MKPSC on node i, $1 \leq i \leq n$

1. *If node i has incorrect $\hat{T}_i$ in Σ_{r-1}, it must update $\hat{T}_i$ in step r.*

2. *For a node i, $\texttt{nowEnter}_i$, $\texttt{requestDelay}_i$ and $\texttt{nowExit}_i$ are pairwise mutual exclusive.*

3. *The membership of node i is changed only by rules* RA *(Exit $\mathcal{S}$ move) and* RC *(Enter $\mathcal{S}$ move). If a node i is privileged to make Exit $\mathcal{S}$ move, it must exit $\mathcal{S}$ successfully under synchronous scheduler (see part(2)).*

4. *If node i exits $\mathcal{S}$, its neighboring nodes can concurrently exits $\mathcal{S}$ if they are eligible to do so.*

5. *A node i can acquire a self-pointer ($\hat{P}_i[0] = i$) only by making Request_Lock move in rule* RA*.* ***Note****: a node cannot acquire a self-pointer by making Update_$\hat{P}$ move in rule* RA*.*

6. *A node i releases its self-pointer when it does not need to enter $\mathcal{S}$ ($\texttt{needEnter}_i = 0$) by making Release_Lock move.*

7. *After a node i with $\texttt{needEnter}_i = 1$ becomes locked, i.e., $\texttt{locked}_i = 1$, it delays its enter*

move by $2k$ steps by making $2k$ Request_Delay moves such that nodes within its distance-k have time to correct their $\hat{T}$ and $\hat{P}$-variables.

8. *Node i either (1) increases d_i by 1 in modulo $2k+1$ by making Request_Delay move (Definitions 4.3.9.3), or (2) clears delay by making Enter or Clear_Delay move such that $d_i = 0$ in Σ_r (Definitions 4.3.9.4 and 4.3.10.3).*

Lemma 4.3.13 *If algorithm MKPSC terminates, then for each node $i \in V$*

(a) If $\hat{T}_i[0] \neq \{i\}$, then $\hat{T}_i[0] = \emptyset$; $\hat{T}_i[\ell] = \bigcup_{j \in N[i]} (\hat{T}_j[\ell - 1])$ for $1 \leq \ell \leq k-1$.

(b) $d_i = 0$.

(c) $\hat{P}_i[\ell] = null$ for $0 \leq \ell \leq k-1$.

(d) $\texttt{nowExit}_i = 0$ and $\texttt{needEnter}_i = 0$.

Proof: (a) This lemma immediately follows from the fact that no node is privileged by rule RA to make Update_$\hat{T}$ move at the termination of the algorithm.

(b) Assume, by contradiction, there exists some node(s) j with $d_j \neq 0$. Node j must have $\texttt{needEnter}_j = 1$ and $\texttt{locked}_j = 1$ (otherwise node j is privileged by rule RA to make Clear_Delay move). Thus, node j is privileged by rule RB or RC, a contradiction.

(c) If no node j has self-pointer (i.e., $\hat{P}_j[0] \neq j$), then $\hat{P}_i[\ell] = null$, $0 \leq \ell \leq k-1$, for all $i \in V$ (otherwise node i is privileged by rule RA to make Update_$\hat{P}$ move). So the key point here is to show that there is no node with self-pointer. Assume, by contradiction, there exists some node(s) with self-pointer. Consider the node with minimum ID from among those nodes, say node i; $\hat{P}_j[\ell] = i$, $1 \leq \ell \leq k-1$, for each node $j \in N[i]$ (otherwise node j would be privileged by the rule RA to make Update_$\hat{P}$ move). Thus, node i is locked (i.e., $\texttt{locked}_i = 1$). Also, node i must have $\texttt{needEnter}_i = 1$ (otherwise node i is privileged by rule RA to make Release_Lock move). Thus, we get node i is privileged by rule RB to make Request_Delay move (by part(b)), a contradiction.

(d) No node i is privileged by rule RA to make Exit $\mathcal{S}$ move and Request_lock move; the claim follows from parts (a) and (c). □

Theorem 4.3.5 *Starting from an arbitrary system state, if algorithm MKPSC terminates using synchronous scheduler, then $\mathcal{S}$ is a maximal k-packing.*

Proof: First, we show $\mathcal{S}$ is a k-packing. Assume, by contradiction, $\mathcal{S}$ is not k-packing, i.e., there exists two $\mathcal{S}$-node(s) i and j such that the length of the shortest path between i and j, $dist(i,j)$, is $\leq k$. Consider node $p \in N(i)$ on the shortest path, it must have $|\hat{T}_p[k-1]| \geq 2$ (Lemma 4.3.13(a)), thus $\texttt{nowExit}_i = 1$; node i is privileged by the rule RA (Exit $\mathcal{S}$), a contradiction. Thus $\mathcal{S}$ is a k-packing.

Next, we claim $\mathcal{S}$ is maximal. Assume otherwise, i.e., there exist some node(s) $i \in \{V - \mathcal{S}\}$ such that $\nexists j \in \mathcal{S} : dist(i,j) \leq k$. Thus $\texttt{needEnter}_i = 1$ and node i is privileged by RA to make Request_Lock move (by Lemma 4.3.13(c)), a contradiction. □

Lemma 4.3.14 *In any system state Σ_r, $r \geq 2k$, if a node i is enabled by the rule* RC *to enter $\mathcal{S}$, (a) no node in $N^k(i)$ is enabled by the rule* RC *to enter $\mathcal{S}$ in system state Σ_{r-k} to Σ_r; (b) each node $j \in N[i]$ must have $|\hat{T}_j[k-1]| \geq |N^{k-1}[j] \cap \mathcal{S}|$.*

Proof: In the system state Σ_{r-2k}, node i must have had $d_i = 0$, $\texttt{needExit}_i = \texttt{locked}_i = 1$; Also, node i made Request_Delay move in each step from $r-2k+1$ to r [otherwise it is impossible for node i to have $d_i = 2k$ in Σ_r (Remark 4.3.5.8 and Request_Delay move)]. Thus, $\texttt{needExit}_i = \texttt{locked}_i = 1$ in system states Σ_{r-2k} to Σ_r.

(a) Node i must be the minimum ID node with self-pointer within its distance-k in Σ_{r-2k}, and no node $j < i$ within distance-k of i got the self-pointer during the steps $[r-2k+1, r-k]$ (otherwise $\texttt{locked}_i$ cannot remain 1 in system state Σ_{r-2k} to Σ_r); Thus $\hat{P}_j[k-1] \neq j$ for all nodes j in $N^k(i)$ in Σ_{r-k}. The lemma holds.

(b) During the steps $[r-k+1, r]$, no node j in $N^k(i)$ can Enter $\mathcal{S}$ by Part(a), and each node $j \in N[i]$ corrected its $\hat{T}_j$ (Remark 4.3.5.1); thus, in Σ_r, each node $j \in N[i]$ must have $|\hat{T}_j[k-1]| \geq |N^{k-1}[j] \cap \mathcal{S}|$). **Note:** it is possible that some node(s) within distance-k of i exits $\mathcal{S}$ during the steps $[r-k+1, r]$ (hence the inequality). □

Lemma 4.3.15 *In step $r \geq 2k+1$, if a node i enters $\mathcal{S}$ (by executing rule* RC*), node i is guaranteed to be the unique $\mathcal{S}$-node within distance-k of node i at the end of current step.*

Proof: If node i enters $\mathcal{S}$ in step r, then node $j \in N[i]$ must have had $|\hat{T}_j[k-1]| = 0$ in Σ_{r-1} (Definition 4.3.9.4), and thus $|N^{k-1}[j] \cap \mathcal{S}| = 0$ by Lemma 4.3.14(b). Coupled with the fact that no other nodes in $j \in N^k(i)$ can enter $\mathcal{S}$ in the same step by Lemma 4.3.14(a), the lemma holds. □

Lemma 4.3.16 *At the end of step $r \geq 2k+1$, if there exists some node(s) $i \in \mathcal{S}$ such that $|N^k[i] \cap \mathcal{S}| \geq 2$, then no nodes in $N^{k-1}[i]$ can make Enter move.*

Proof: This lemma immediately follows from Lemma 4.3.15. □

Theorem 4.3.6 *Starting from any initial illegitimate state, algorithm `MKPSC` converges to a safe state ($\mathcal{S}$ denotes a k-packing, i.e, each node $i \in \mathcal{S}$ is the unique $\mathcal{S}$-node within distance-k of node i) after $3k+1$ steps.*

Proof: Consider any node $i \in \mathcal{S}$ with $|N^k[i] \cap \mathcal{S}| \geq 2$ in Σ_{2k+1}: $\hat{T}_j[k-1]$ on $j \in N(i)$ must be ≥ 2 in Σ_{3k} by Lemma 4.3.16 and Remark 4.3.5.1, thus node i has $\texttt{nowExit}_i = 1$ in Σ_{3k}. In the step $3k+1$, i must exit $\mathcal{S}$ by executing the rule RA (Remark 4.3.5.3). Thus all nodes $i \in \mathcal{S}$ with $|N^k[i] \cap \mathcal{S}| \geq 2$ in Σ_{2k+1} will be out of $\mathcal{S}$ at the end of step $3k+1$. Coupled with the fact that each newly created $\mathcal{S}$-node must be the the unique $\mathcal{S}$-node within its distance-k (Lemma 4.3.15), the lemma holds. □

Theorem 4.3.7 *After step $3k+1$, algorithm `MKPSC` maintains safety in all subsequent steps before converging to a legitimate state.*

Proof: After step $3k+1$, we reach a safe state. If any node i enters $\mathcal{S}$, then it is guaranteed to be the unique $\mathcal{S}$-node within distance-k of node i by Lemma 4.3.15; thus we reach another safe state. □

Lemma 4.3.17 *Starting from a safe state Σ_r, $r \geq 4k$, no node will ever make Exit $\mathcal{S}$ move in subsequent steps.*

Proof: In a safe state, there are no two $\mathcal{S}$-nodes i and j such that $dist(i,j) \leq k$. The algorithm `MKPSC` always in the safe state after step $3k+1$ (Theorem 4.3.7), thus each node i always has $|N^{k-1}[i] \cap \mathcal{S}| \leq 1$ after step $3k+1$ and each node i always has $\hat{T}_i[k-1] \leq 1$ after step $4k$ by Remark 4.3.5.1. The lemma holds. □

Lemma 4.3.18 *Starting from a safe (not legitimate) state Σ_r, $r \geq 4k$, the number of $\mathcal{S}$-node increases in at most $k(n+4)+2$ steps.*

Proof: (a) **No node makes Update_$\hat{T}$ moves after step** $r+k$: this is true because there is no Exit move after step $4k$ (Lemma 4.3.17) and each node i corrects its $\hat{T}_i$ during the steps $[r+1, r+k]$.

(b) **No node makes Release_Lock move after step** $r + k + 1$: each node i has correct $\hat{T}_i$ at the end of step $r + k$ (part(a)). After another step, all nodes i with $\texttt{needEnter}_i = 0$ release locks simultaneously.

(c) **No node makes Request_Lock and Update_$\hat{P}$ moves after step** $r + k(n + 2) + 1$: After step $r + k + 1$, no node makes Release_Lock move, thus the number of nodes with self-pointer is non-decreasing. Each Request_Lock move increases the number of nodes with self-pointer by 1, so there are at most n Request_Lock moves after step $r + k + 1$. In between any two consecutive Request_Lock moves, there are at most k Update_$\hat{P}$ moves.

(d) **No node makes Request_Delay and Clear_Delay moves after step** $r+k(n+4)+1$: After step $r + k(n + 2) + 1$, Request_Delay and Clear_Delay moves can be made in at most $2k$ subsequent steps by parts (a), (b) and (c).

Thus, at least one node makes Enter move in step $r + k(n + 4) + 2$. □

Theorem 4.3.8 *Starting at an arbitrary state, algorithm* `MKPSC` *terminates in* $O(kn^2)$ *steps under the synchronous scheduler.*

Proof: After step $4k$, the number of $\mathcal{S}$-nodes is non-decreasing by Lemma 4.3.17, and the number of $\mathcal{S}$-nodes increases in at most $k(n + 4) + 2 = O(kn)$ steps by Lemma 4.3.18. Thus, the algorithm terminates in $4k + n \times O(kn) = O(kn^2)$ steps. □

Chapter 5

Experimental Verification and Performance Study

5.1 Simulation Environmental Settings

Simulation studies are used to verify the correctness and efficiencies of the algorithms proposed in this thesis. The performance of an algorithm is evaluated in terms of both convergence time (i.e., the number of steps during convergence) and the size of $\mathcal{S}$ in various topologies. Random graphs are generated to evaluate the performance of algorithms `MWCDS-S` and `M2PSC`, while six social network graphs (the dataset is available at http://snap.stanford.edu/ data/egonets-Facebook.html) are retrieved from Facebook to evaluate the performance of algorithm `MWPIDS1`. In addition, the behaviors of algorithm `MFGASC` are simulated and analyzed by specifying different values for parameters f and g at each node of the network graph.

5.1.1 Network Graph Generation

The network graphs for experiments are generated using the ad hoc network graph generation model in [29], which builds a random connected graph G_n with n nodes in incremental fashion. As the starting point, G_1 includes exactly one node positioned at $(0, 0)$ in Cartesian coordinates. Given G_{k-1} with $k-1$ nodes ($k \geq 2$), G_k is generated as follows:

1. Choose an existing node i randomly from G_{k-1} using the geometric distribution:

$$Pr(i) = \gamma(1-\gamma)^{\delta_i - 1}$$

where $0 < \gamma \leq 1$ and δ_i is the degree of node i. Note that larger value of γ gives a graph with smaller average node degree.

2. A new node j is placed randomly in coverage area of i at polar coordinates (r, θ) with $r = \alpha R$ and $0 \leq \theta \leq 2\beta\pi$, where R is the radius of the coverage area of node i and $0 < \alpha, \beta \leq 1$. Note that: (a) the value of R does not affect the topology of the network graph; (b) larger value of α gives a more geographically spread-out graph; and (c) smaller value of β gives a higher diameter graph.

3. Add an edge between node p in G_{k-1} to j if the Cartesian distance between nodes p and j is at most R.

Figure 5.1 shows two sample graphs generated using the ad hoc network graph model above. Both graphs include 50 nodes; while Figure 5.1(b) is more geographically spread-out and has smaller average node degree and diameter than Figure 5.1(a) due to the larger values of parameters α, β, and γ.

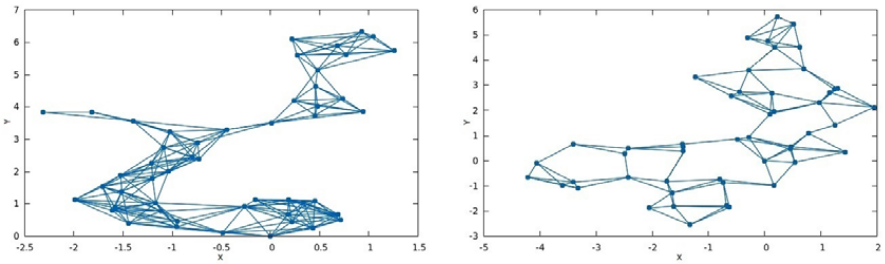

Figure 5.1: Example graphs with $n = 50$

5.1.2 Initial Global State Generation

In a self-stabilizing algorithm, each node maintains a set of local variables that defines the local state of the node. The union of the local states of all nodes in the system is called the global state. The execution of self-stabilizing algorithm starts with an initial global state. In the experiments, the initial global state is generated by giving random (but valid) value for each local

variable of nodes in the network graph. For instance, algorithm M2PSC requires that each node $i \in V$ maintains four variables: $\boldsymbol{s_i}$, $\boldsymbol{c_i}$, $\boldsymbol{p_i}$, and $\boldsymbol{d_i}$. The values of variables s_i and d_i are randomly chosen from {true, false} as they both are boolean flags. The randomly generated value of variable c_i must be nonnegative as algorithm M2PSC requires c_i to be nonnegative. The value of pointer p_i can be an arbitrary integer: if $p_i \leq 0$, then node i points null, otherwise it points a node numbered by the value of p_i.

5.1.3 Runtime Scheduler Implementation

In any global state, there may exist multiple privileged nodes. A runtime scheduler (also called a daemon) is assumed to select the privileged node(s) to move. The most common schedulers include synchronous scheduler, central scheduler, and distributed scheduler. The implementation of a synchronous scheduler is trivial as all the privileged nodes are selected to move at each step; we simply let each privileged node to move. A central scheduler selects exactly one privileged node to move at each step; if k nodes are privileged in a global state, then we first determine an integer p uniformly at random in the range of $[1, k]$, and then select p-th node in the privileged nodes list to move. A distributed scheduler selects a non-empty subset of the privileged nodes to move at each step; thus, for any system state with k privileged nodes, an integer p is first determine uniformly at random in the range of $[1, k]$, and then p nodes among from all k privileged nodes are randomly chosen to move.

5.2 Case Studies

5.2.1 Minimal Weakly Connected Dominating Set Algorithm

Using the ad hoc network graph generation model in section 5.1.1, the performance of algorithm MWCDS-S is evaluated in terms of both convergence time (i.e., the number of steps during convergence) and the size of computed minimal weakly connected dominating set in various topologies. The local state of each node (the values of s, d and m-variables) and root node are given randomly.

The size of graph in the experiments varies from 100 to 1000. R remains 1 throughout all tests as the value of R does not affect the topology of the generated graph (and hence does not

affect the convergence time and size of $\mathcal{S}$). The experimental results show that algorithm MWCDS-S computes a minimal weakly connected dominating set for each tested graph. Figure 5.2 shows the changes of convergence time and size of $\mathcal{S}$ in different size of graphs generated using different values of parameters α, β and γ.

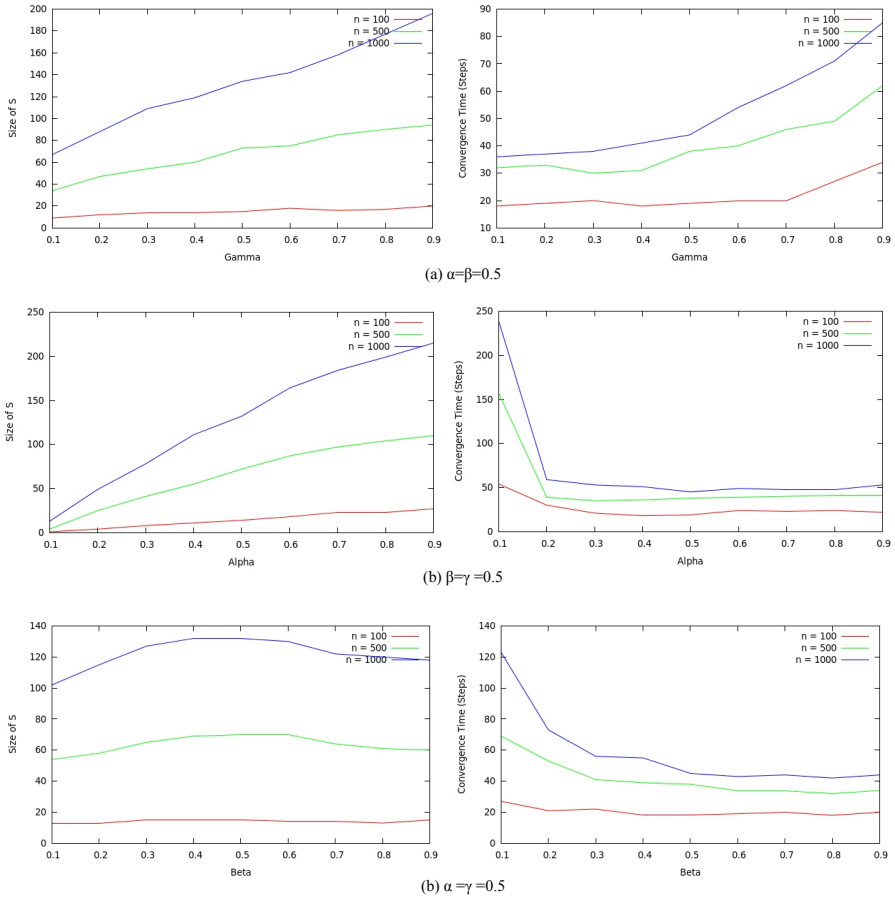

Figure 5.2: Experimental results using algorithm MWCDS-S

In Figure 5.2(a), the values of α and β remain 0.5. As the value of γ increases, (1) the size of $\mathcal{S}$ increases: the larger value of γ results in a graph with smaller average node degree, hence $\mathcal{S}$ includes more nodes in order to obtain the connectivity and domination properties of MWCDS; (2) the convergence time increases: the larger value of γ tends to give a larger diameter graph, which increases the time for algorithm MWCDS-S to reach the d-legitimate state.

In Figure 5.2(b), the values of β and γ remain 0.5. As the value of α increases, (1) the size of $\mathcal{S}$ increases: the larger value of α results in a more spread-out graph with smaller average node

degree, hence $\mathcal{S}$ includes more nodes in order to obtain the connectivity and domination properties of MWCDS; (2) the convergence time first rapidly decreases and then increases: when $\alpha = 0.1$, most of nodes in graph are connected to each other and have the same distance to the selected root node, and hence compete to change their memberships simultaneously. Algorithm `MWCDS-S` forces the nodes with the same distance to root to change their memberships in sequential (implemented by stored m-variables), which results in large convergence time. As α increases to 0.2, the generated graph becomes more spread-out, hence the convergence time decreases rapidly. After that, the generated graph continues becoming more spread-out, which results in a larger diameter and hence increases the time for algorithm `MWCDS-S` to reach the d-legitimate state; this explains why the convergence time increases when α keeps increasing.

In Figure 5.2(c), the values of α and γ remain 0.5. As the value of β increases, (1) the convergence time decreases: the larger value of β gives a smaller diameter graph, which decreases the time for algorithm `MWCDS-S` to reach the d-legitimate state; (2) the size of $\mathcal{S}$ remains at a relatively stable level.

Figures 5.2(a), (b) and (c) also show that, for tested ad hoc network graphs, (1) on average the size of $\mathcal{S}$ is around $0.12n$ and (2) algorithm `MWCDS-S` terminates within n steps, where n is the number of nodes in the network graph.

5.2.2 Influential Users Selection Algorithm in Social Networks

To evaluate the performance of algorithm `MWPIDS1`, six social network graphs are retrieved from Facebook (the dataset is available at http://snap.stanford.edu/data/egonets-Facebook.html). Note that the isolated nodes in the graph represent the individuals without any friends in the real social network; they do not contribute to the influence propagation in the social network. Thus, these social network graphs are preprocessed to eliminate the isolated nodes. The resulting social network graphs are shown in Table 5.1.

Each edge (i, j) in the social network graph is viewed as two parallel directed edges $i \to j$ and $j \to i$. For each directed edge $i \to j$, the weight $w_{i,j}$ is generated uniformly at random in the range of $[0, 10]$, where $w_{i,j}$ indicates the degree of influence of i on j (e.g., i has no influence on j if $w_{i,j} = 0$); the value of t_i denoting the level of tolerance (or sensitivity) of i is generated uniformly at random in the range of $[0, \gamma \sum_{j \in N(i)} w_{j,i}]$, where $0 < \gamma \leq 1$. Note that the larger values of γ result in low-sensitivities of nodes, i.e., nodes require more influence to be convinced about the same

Table 5.1: Dataset description

Graph ID	Graph size (n)	Number of edges (m)	Avg. degree of nodes
1	61	540	17.70
2	150	3386	45.15
3	224	6384	57.00
4	333	5038	30.26
5	534	9626	36.05
6	786	28048	71.37

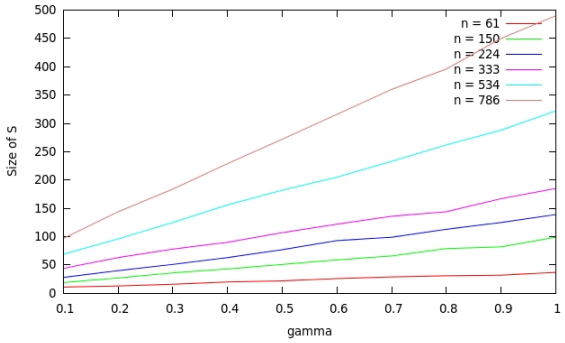

Figure 5.3: The impact of γ on the size of generated MWPIDS

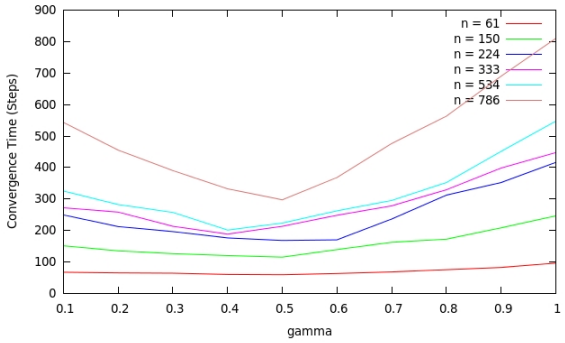

Figure 5.4: The impact of γ on the convergence time

opinion or tend to show the same behavior as their neighbors.

Algorithm `MWPIDS1` is tested on six social network graphs as shown in Table 5.1. The convergence time is measured in terms of steps, and the size of generated minimal weighted positive influence dominating set (i.e., $|\mathcal{S}|$) is computed for each test. The experimental results show that algorithm `MWPIDS1` computes a minimal weighted positive influence dominating set for each tested graph. Figures 5.3 and 5.4 show the variations in convergence time and size of $\mathcal{S}$ in each tested graph using different values of parameter γ.

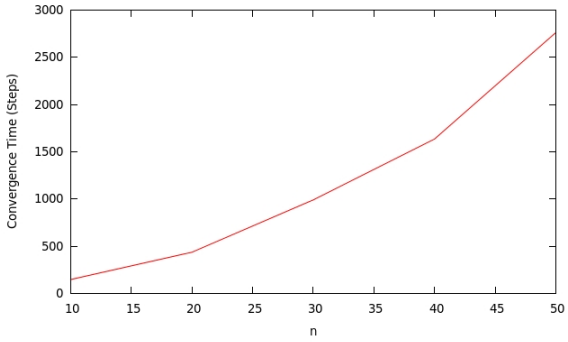

Figure 5.5: The impact of scheduler on the convergence time ($\gamma = 1$)

Figure 5.3 shows that: (1) *As the value of γ increases, the size of $\mathcal{S}$ increases:* the larger value of γ results in low-sensitivities of nodes (i.e., nodes require more influence to be convinced about the same opinion or tend to show the same behavior as their neighbors); hence $\mathcal{S}$ includes more nodes in order to ensure each node in the graph obtains enough influence from its $\mathcal{S}$-neighbors. (2) *For the same value of γ, as the size of graph increases, the size of computed MWPIDS also increases.*

Figure 5.4 shows that: (1) *As the value of γ increases, the convergence time first decreases and then increases:* in the initial state, the expected number of $\mathcal{S}$-nodes is $n/2$ for a graph of size n as the value of s-variable at each node is given uniformly at random. When $\gamma = 0.1$, nodes in the graph tend to have relatively small t-values; the $\mathcal{S}$-nodes compete to exit $\mathcal{S}$ to achieve minimality of $\mathcal{S}$, which results in large convergence time. As γ increases to 0.5, t-value at each node increases, thus the number of $\mathcal{S}$ nodes needs to exit decreases, and further the convergence time decreases. After that, t-values continues becoming larger, which results in non-$\mathcal{S}$-nodes compete to enter $\mathcal{S}$ in order to ensure the legalities of nodes (Definition 3.2.1), this explains why the convergence time increases when γ keeps increasing. (2) *For the same value of γ, as the size of graph increases, the convergence time also increases.* For each tested social network graph with n nodes, algorithm `MWPIDS1` terminates within $1.2n$ steps. However, it is to be noted that the behavior of the scheduler also affect the convergence time, e.g, if the scheduler is forced to select exactly one node to move in each step, then the convergence time may increase to around n^2 steps for complete graph with n nodes as shown in Figure 5.5.

5.2.3 Maximal 2-packing Algorithm

Using the ad hoc network graph generation model in section 5.1.1, the performance of algorithm M2KSC is evaluated in terms of both convergence time (i.e., the number of steps during convergence) and the size of computed maximal 2-packing in various topologies. The local state of each node (the values of s, c, p and d-variables) is given randomly.

The size of graph in the experiments varies from 100 to 1000. R remains 1 throughout all tests as the value of R does not affect the topology of the generated graph (and hence does not affect the convergence time and size of $\mathcal{S}$). The experimental results show that algorithm M2KSC computes a 2-packing in at most 3 steps, and then terminates in a maximal one for each tested graph. Figure 5.6 shows the changes of convergence time and size of $\mathcal{S}$ in different size of graphs generated using different values of parameters α, β and γ.

In Figure 5.6(a), the values of α and β remain 0.5. As the value of γ increases, (1) the size of $\mathcal{S}$ increases: the larger value of γ results in a graph with smaller average node degree, hence $\mathcal{S}$ includes more nodes in order to obtain the domination property (i.e., each non-$\mathcal{S}$-node is dominated by at least one $\mathcal{S}$-node in its distance-2 neighborhood); (2) the convergence time increases: the size of $\mathcal{S}$ increases (as discussed in part(1)), hence more nodes need to enter $\mathcal{S}$ by executing Enter moves, which increases the time for algorithm M2KSC to reach the legitimate state.

In Figure 5.6(b), the values of β and γ remain 0.5. As the value of α increases, (1) the size of $\mathcal{S}$ increases: the larger value of α results in a more spread-out graph with smaller average node degree, hence $\mathcal{S}$ includes more nodes in order to obtain the domination property; (2) the convergence time first increases rapidly and then slowly: when $\alpha = 0.1$, most of nodes in graph are connected to each other; as α increases to 0.2, the generated graph becomes more spread-out and the average node degree deceases rapidly, thus more nodes need to enter $\mathcal{S}$ by executing Enter moves in order to obtain the domination property, which results the rapid increasing of the convergence time. After that, the generated graph continues becoming more spread-out but the average node degree remains at a relatively stable level, this explains why the convergence time increases slowly when α keeps increasing.

In Figure 5.6(c), the values of α and γ remain 0.5. As the value of β increases, (1) the convergence time decreases: the larger value of β gives a smaller diameter graph where nodes are distributed more evenly, thus multiple nodes tend to make Enter moves simultaneously, which

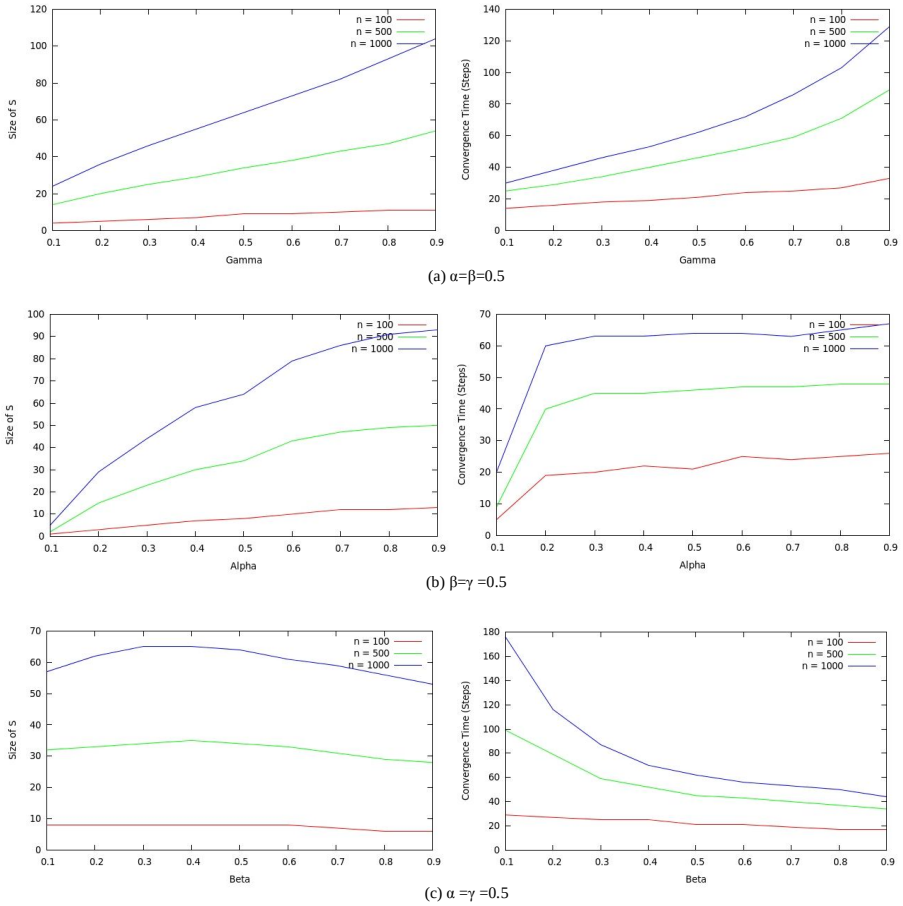

Figure 5.6: Experimental results using algorithm M2KSC

decreases the time for algorithm M2KSC to reach the legitimate state.

Figures 5.6(a), (b) and (c) also show that, for tested ad hoc network graphs, (1) on average the size of $\mathcal{S}$ is around $0.06n$, and (2) algorithm M2KSC terminates within n steps, where n is the number of nodes in the network graph. It is to be noted that each non-$\mathcal{S}$-node is dominated by at least one $\mathcal{S}$-node in its distance-1 neighborhood for MWCDS, while each non-$\mathcal{S}$ node is dominated by at least one $\mathcal{S}$-node in its distance-2 neighborhood for maximal 2-packing; this explains why the size of computed minimal weakly connected dominating set is larger than that of maximal 2-packing as shown in Figures 5.2 and 5.6.

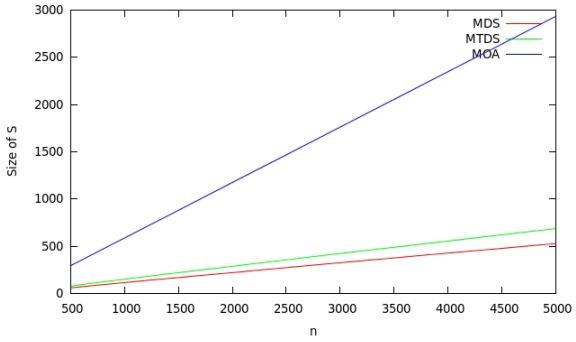

Figure 5.7: The impact of parameters f and g on the size of $\mathcal{S}$

5.2.4 Minimal $(f, g)-$Alliance Algorithm

Authors in [19] have observed that the concept of minimal (f, g)-alliance actually generalizes some already existing graph theoretical invariants. Specifically, a minimal (f, g)-alliance is:

(1) a minimal dominating set iff $f_i = 1$ and $g_i = 0$ for each $i \in V$;

(2) a minimal total dominating set iff $f_i = g_i = 1$ for each $i \in V$; and

(3) a minimal offensive alliance iff $f_i = \lceil \delta_i/2 \rceil$ and $g_i = 0$ for each $i \in V$.

Using the ad hoc network graph generation model in section 5.1.1, the performance of algorithm `MFGASC` is evaluated in terms of both convergence time and the size of computed alliance. We first study the effects of graph topology on convergence time and size of $\mathcal{S}$ using different values of parameters α, β and γ (as we do for algorithm `M2KSC`). The experimental results show that algorithm `MFGASC` computes a (f, g)-alliance in at most 3 steps, and then terminates in a maximal one within n steps for each tested graph with n nodes. Also, the effects of graph topology on the convergence time and size of $\mathcal{S}$ for algorithm `MFGASC` are similar to those for algorithm `M2KSC` (we omit the details).

Here we focus on the effects of parameters f and g on convergence time and size of $\mathcal{S}$ for different graph invariants, i.e., minimal dominating set (MDS), minimal total dominating set (MTDS), and minimal offensive alliance (MOA). The size of graph in the experiments varies from 500 to 5000. $\alpha = \beta = \gamma = 0.5$ and R remains 1 throughout all tests. The experimental results show that algorithm `MFGASC` computes a minimal (f, g)-alliance for each tested graph.

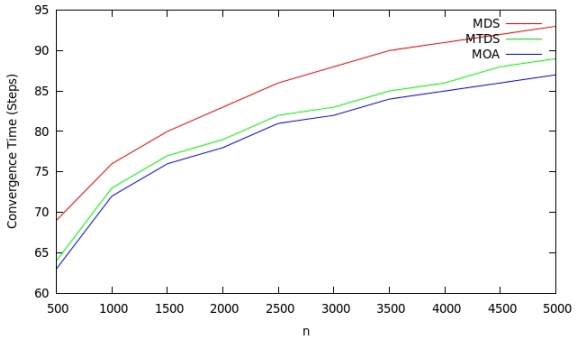

Figure 5.8: The impact of parameters f and g on the convergence time

Figure 5.7 shows that: (1) the size of $\mathcal{S}$ increases as the number of nodes in the graph (i.e., n) increases; (2) for the same graph, the sizes of $\mathcal{S}$'s computed by algorithm `MFGASC` for MOA, MTDS and MDS are in descending order, which is consistent with the domination properties specified by the values of parameters f and g.

Figure 5.8 shows that: (1) the size of convergence time increases as the number of nodes in the graph (i.e., n) increases; (2) the convergence times for algorithm `MFGASC` to compute MOA, MTDS and MDS in the same graph are in increasing order: algorithm `MFGASC` reaches to a safe state in at most 3 steps (note: most of nodes in the graph belong to $\mathcal{S}$ in such safe state according to the simulation); after that, nodes can only Exit $\mathcal{S}$, i.e., no node will ever Enter $\mathcal{S}$ move; thus the number of nodes exiting $\mathcal{S}$ for algorithm `MFGASC` to compute MOA, MTDS and MDS are in increasing order, so do the convergence times.

Chapter 6

Conclusion

6.1 Contribution Summary

In this thesis, a survey of self-stabilizing algorithms for classical graph theoretic invariants is first provided. Then, a collection of new self-stabilizing algorithms for two variants of the dominating set (i.e., minimal weakly connected dominating set and minimal weighted positive influence dominating set) are proposed. At last, three safe converging self-stabilizing algorithms are proposed for graph alliance and packing problems. Specifically, the contributions are four-folds as follows:

1. Three new self-stabilizing algorithms are proposed to compute minimal weakly connected dominating sets in an arbitrary connected graph:

   - Algorithm `MWCDS-S` terminates in $O(n)$ steps using a synchronous scheduler; it uses a distinguished root node and assumes unique node IDs;
   - Algorithm `MWCDS-C` stabilizes in $O(n^4)$ steps using an unfair central scheduler; it uses a distinguished root node while other nodes are anonymous;
   - Algorithm `MWCDS-D` stabilizes in $O(n^4)$ steps using an unfair distributed scheduler; it uses a distinguished root node and assumes unique node IDs;

   where n is the number of nodes in the network graph; note that they all assume the existence of a distinguished root node and have $O(\log n)$ bits of space requirement at each node.

2. The users in the minimal weighted positive influence dominating set of a social network graph are selected as the influential users in this thesis. A new self-stabilizing algorithm is presented to compute a minimal weighted positive influence dominating set for an arbitrary social network. Using a distributed scheduler, it terminates in $O(n^3)$ steps, where n is the number of nodes in the network graph. Space requirement at each node is $O(\log n)$ bits.

3. A new self-stabilizing algorithm with safe convergence is proposed for minimal (f, g)-alliance problem. Using a synchronous scheduler, it quickly converges to an (f, g)-alliance in at most three steps, and then terminates in a minimal one in $O(n)$ steps without breaking safety during the convergence interval, where n is the number of nodes in the network graph. Space requirement at each node is $O(\log n)$ bits.

4. The first self-stabilizing maximal 2-packing algorithm with safe convergence is proposed. Assuming a synchronous scheduler, for an arbitrary graph with n nodes, the proposed algorithm first converges to a 2-packing in at most three steps, and then converges to a maximal one in $O(n)$ steps without breaking safety rule during the stabilization interval. Space requirement at each node is $O(\log n)$ bits. The technique in maximal 2-packing algorithm is then generalized to design a self-stabilizing algorithm for maximal k-packing, $k \geq 2$, with safe convergence that stabilizes in $O(kn^2)$ steps under a synchronous scheduler; the algorithm has space complexity of $O(kn \log n)$ bits at each node.

6.2 Future Work

The following interesting problems are proposed as the future work:

1. *Self-stabilizing minimal connected dominating set algorithm*: three self-stabilizing algorithms are proposed for minimal weakly connected dominating set, but there does not exist any self-stabilizing minimal connected dominating set algorithm in the literature. It would be interesting to investigate if it is possible to propose self-stabilizing algorithms for computing minimal connected dominating set.

2. *Linear time self-stabilizing algorithm for maximal k-packing with safe convergence*: a straightforward application of the proposed maximal k-packing algorithm (in chapter 4) to the case of $k = 2$ will result in a self-stabilizing algorithm of $O(kn^2)$ time complexity, which is not as

good as the proposed maximal 2-packing algorithm. It is then an open problem if one can design an $O(n)$ self-stabilizing algorithm for maximal k-packing with safe convergence.

Bibliography

[1] Amazon. Amazon S3 availability event: July 20, 2008.

[2] F. Bonchi. Influence propagation in social networks: a data mining perspective. *IEEE Intelligent Informatics Bulletin*, 12:8–16, 2011.

[3] F. Carrier, A.K. Datta, S. Devismes, L.L. Larmore, and Y. Rivierre. Self-stabilizing (f, g)-alliances with safe convergence. In *15th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, Osaka, Japan, Nov. 2013.

[4] W. Chen, L.V.S. Lakshmanan, and C. Castillo. *Information and Influence Propagation in Social Networks*. Morgan & Claypool, 2013.

[5] W. Chen, C. Wang, and Y. Wang. Scalable influence maximization for prevalent viral marketing in large-scale social networks. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1029–1038, 2010.

[6] Y.P. Chen and A.L. Liestman. Approximating minimum size weakly-connected dominating sets for clustering mobile ad hoc networks. In *Proceedings of the 3rd ACM international symposium on Mobile ad hoc networking and computing*, pages 165–172, 2002.

[7] J. Deng, Y.S. Han, W.B. Heinzelman, and P.K. Varshney. Scheduling sleeping nodes in high density cluster-based sensor networks. In *ACM/Kluwer Mobile Networks and Applications Special Issue on Energy Constraints and Lifetime Performance in Wireless Sensor Networks*, pages 825–835, 2004.

[8] E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, Nov. 1974.

[9] Y. Ding, J.Z. Wang, and P.K. Srimani. Self-stabilizing algorithm for minimal global powerful alliance with safe convergence in an arbitrary graph. *Journal of Computer and System Sciences*, 2013. Submitted.

[10] Y. Ding, J.Z. Wang, and P.K. Srimani. A linear time self-stabilizing algorithm for minimal weakly connected dominating sets. *International Journal of Parallel Programming*, 2014.

[11] Y. Ding, J.Z. Wang, and P.K. Srimani. New self-stabilizing algorithms for minimal weakly connected dominating sets. *International Journal of Foundations of Computer Science*, 2014.

[12] Y. Ding, J.Z. Wang, and P.K. Srimani. Polynomial time self-stabilizing algorithm for minimal total dominating set in arbitrary graphs. *Journal of Applied Mathematics and Computing*, 2014. Submitted.

[13] Y. Ding, J.Z. Wang, and P.K. Srimani. Self-stabilizing algorithm for maximal 2-packing with safe convergence in an arbitrary graph. In *28th IEEE IPDPS Workshop on Advances in Parallel and Distributed Computational Models*, 2014.

[14] Y. Ding, J.Z. Wang, and P.K. Srimani. Self-stabilizing algorithm for minimal dominating set with safe convergence in an arbitrary graph. *Parallel Processing Letters*, 2014.

[15] Y. Ding, J.Z. Wang, and P.K. Srimani. Self-stabilizing minimal global offensive alliance algorithm with safe convergence in an arbitrary graph. In *11th Annual Conference on Theory and Applications of Models of Computation*, April 2014.

[16] Y. Ding, J.Z. Wang, and P.K. Srimani. Self-stabilizing selection of influential users in social networks. In *13th International Symposium on Pervasive Systems, Algorithms, and Networks*, 2014.

[17] T.N. Dinh, Y. Shen, D.T. Nguyen, and M.T. Thai. On the approximability of positive influence dominating set in social networks. *Journal of Combinatorial Optimization*, 27:487–503, 2014.

[18] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7(1):3–16, Nov. 1993.

[19] M.C. Dourado, L.D. Penso, D. Rautenbach, and J.L. Szwarcfiter. The south zone: distributed algorithms for alliances. In *Proceedings of the 13th international conference on Stabilization, safety, and security of distributed systems*, pages 178–192, 2011.

[20] M. Gairing, R.M. Geist, S.T. Hedetniemi, and P. Kristiansen. A self-stabilizing algorithm for maximal 2-packing. *Nordic Journal of Computing*, 11:1–11, 2004.

[21] M. Gairing, S.T. Hedetniemi, P. Kristiansen, and A.A. McRae. Distance-two information in self-stabilizing algorithms. *Parallel Processing Letters*, 14:387–398, 2004.

[22] R. Gallant, G. Gunther, B. Hartnell, and D.F. Rall. Limited packing in graphs. *Discrete Applied Mathematics*, 158:1357–1364, 2010.

[23] Gmail. Gmail disaster: Reprots of mass email deletions: December 28, 2006.

[24] W. Goddard, S.T. Hedetniemi, D.P. Jacobs, and P.K. Srimani. A self-stabilizing distributed algorithm for minimal total domination in an arbitrary system graph. In *Proceedings of 8th IPDPS Workshop on Formal Methods for Parallel Programming*, pages 240–243, April 2003.

[25] W. Goddard, S.T. Hedetniemi, D.P. Jacobs, and P.K. Srimani. Self-stabilizing global optimization algorithms for large network graphs. *International Journal of Distributed Sensor Networks*, 1(3-4):329–344, 2005.

[26] W. Goddard, S.T. Hedetniemi, D.P. Jacobs, P.K. Srimani, and Z. Xu. Self-stabilizing graph protocols. *Parallel Processing Letters*, 18:189–199, 2008.

[27] W. Goddard and P.K. Srimani. Daemon conversions in distributed self-stabilizing algorithms. In *WALCOM: Algorithms and computation*, pages 146–157, 2013.

[28] N. Guellati and H. Kheddouci. A survey on self-stabilizing algorithms for independence, domination, coloring, and matching in graphs. *Journal of Parallel and Distributed Computing*, 70(4):406–415, 2010.

[29] S.K.S. Gupta and P.K. Srimani. Adaptive core selection and migration method for multi-cast routing in mobile ad hoc networks. *IEEE Transactions on Parallel and Distributed Systems*, 14:27–38, 2003.

[30] S.M. Hedetniemi, S.T. Hedetniemi, D.P. Jacobs, and P.K. Srimani. Self-stabilizing algorithms for minimal dominating sets and maximal independent sets. *Computer Mathematics and Applications*, 46:805–811, 2003.

[31] S.T. Hedetniemi, D.P. Jacobs, and V. Trevisan. Distance-k knowledge in self-stabilizing algorithms. *Theoretical Computer Science*, 339:118–127, 2008.

[32] H. Kakugawa and T. Masuzawa. A self-stabilizing minimal dominating set algorithm with safe convergence. In *20th IEEE International Parallel and Distributed Processing Symposium*, pages 25–29, 2006.

[33] S. Kamei and H. Kakugawa. A self-stabilizing approximation algorithm for the minimum weakly connected dominating set with safe convergence. In *Proceedings of the 1st International Workshop on Reliability, Availability, and Security*, 2007.

[34] S. Kamei and H. Kakugawa. A self-stabilizing approximation for the minimum connected dominating set with safe convergence. In *Proceedings of the 12th International Conference on Principles of Distributed Systems*, pages 496–511, Luxor, Egypt, 2008. Springer-Verlag.

[35] S. Kamei and H. Kakugawa. A self-stabilizing distributed approximation algorithm for the minimum connected dominating set. *International Journal of Foundations of Computer Science*, 21:459–476, 2010.

[36] S. Kamei and H. Kakugawa. A self-stabilizing 6-approximation for the minimum connected dominating set with safe convergence in unit disk graphs. *Theoretical Computer Science*, 428:80–90, April 2012.

[37] D. Kempe, J. Kleinberg, and E. Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 137–146, 2003.

[38] M. Kimura, K. Saito, and R. Nakano. Extracting influential nodes for information diffusion on a social network. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence*, pages 1371–1376, 2007.

[39] F. Manne and M. Mjelde. A memory efficient self-stabilizing algorithm for maximal k-packing. In *Proceedings of the 8th International Conference on Stabilization, Safety, and Security of Distributed Systems*, pages 428–439, Berlin, Heidelberg, 2006.

[40] H. Raei, N. Yazdani, and M. Asadpour. A new algorithm for positive influence dominating set in social networks. In *2012 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pages 253–257, 2012.

[41] Z. Shi. An updated self-stabilizing algorithm to maximal 2-packing and a linear variation under synchronous daemon. In *2011 International Conference on Parallel and Distributed Processing Techniques and Applications*, volume 1, pages 262–267, 2011.

[42] Z. Shi. A self-stabilizing algorithm to maximal 2-packing with improved complexity. *Information Processing Letters*, 112:525–531, Jul. 2012.

[43] P.K. Srimani and Z. Xu. Distributed protocols for defensive and offensive alliances in network graphs using self-stabilization. In *International Conference on Computing: Theory and Applications*, pages 27–31, Kolkata, Mar. 2007.

[44] P.K. Srimani and Z. Xu. Self-stabilizing algorithms of constructing spanning tree and weakly connected minimal dominating set. In *27th International Conference on Distributed Computing Systems Workshops*, 2007.

[45] J. Tang, J. Sun, C. Wang, and Z. Yang. Social influence analysis in large-scale networks. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 807–816, 2009.

[46] V. Turau. Linear self-stabilizing algorithms for the independent and dominating set problems using an unfair distributed scheduler. *Information Processing Letters*, 103(3):88–93, July 2007.

[47] V. Turau. Efficient transformation of distance-2 self-stabilizing algorithms. *Journal of Parallel and Distributed Computing*, 72:603–612, 2012.

[48] V. Turau and B. Hauck. A self-stabilizing algorithm for constructing weakly connected minimal dominating sets. *Information Processing Letters*, 109:763–767, 2009.

[49] F. Wang, E. Camacho, and K. Xu. Positive influence dominating set in online social networks. In *Proceedings of the 3rd International Conference on Combinatorial Optimization and Applications*, pages 313–321, 2009.

[50] F. Wang, H. Du, E. Camacho, K. Xu, W. Lee, Y. Shi, and S. Shan. On positive influence dominating sets in social networks. *Theoretical Computer Science*, 412:265–269, 2011.

[51] G. Wang, H. Wang, X. Tao, and J. Zhang. A self-stabilizing algorithm for finding a minimal positive influence dominating set in social networks. In *Proceedings of the Twenty-Fourth Australasian Database Conference*, pages 93–99, 2013.

[52] Z. Xu, S.T. Hedetniemi, W. Goddard, and P.K. Srimani. A synchronous self-stabilizing minimal domination protocol in an arbitrary network graph. In *Proceedings of the 5th International Workshop on Distributed Computing*, pages 26–32, Calcutto, India, Dec. 2003.

[53] Z. Xu, J. Wang, and P.K. Srimani. Distributed fault tolerant computation of weakly connected dominating set in ad hoc networks. *The Journal of Supercomputing*, 53:182–195, 2010.

[54] S. Yahiaoui, Y. Belhoul, M. Haddad, and H. Kheddouci. Self-stabilizing algorithms for minimal global powerful alliance sets in graphs. *Information Processing Letters*, 113:365–370, 2013.

[55] H. Zhang, T.N. Dinh, and M.T. Thai. Maximizing the spread of positive influence in online social networks. In *IEEE 33rd International Conference on Distributed Computing Systems*, pages 317–326, 2013.

Author's Publications

[1] Y. Ding, J.Z. Wang, and P.K. Srimani. Churn tolerance algorithm for state machine replication. In *2012 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, volume 2, pages 356–360, Dec. 2012.

[2] Y. Ding, J.Z. Wang, and P.K. Srimani. Self stabilizing master-slave token circulation algorithm in an undirected ring of arbitrary size and its orientation. In *27th IEEE IPDPS Workshop on Advances in Parallel and Distributed Computational Models*, pages 659–666, 2013.

[3] Y. Ding, J.Z. Wang, and P.K. Srimani. Fault-tolerant distributed publish/subscribe using self-stabilization. In *6th International Symposium on Parallel Architectures, Algorithms and Programming*, 2014.

[4] Y. Ding, J.Z. Wang, and P.K. Srimani. A linear time self-stabilizing algorithm for minimal weakly connected dominating sets. *International Journal of Parallel Programming*, 2014.

[5] Y. Ding, J.Z. Wang, and P.K. Srimani. New self-stabilizing algorithms for minimal weakly connected dominating sets. *International Journal of Foundations of Computer Science*, 2014.

[6] Y. Ding, J.Z. Wang, and P.K. Srimani. Self-stabilizing algorithm for maximal 2-packing with safe convergence in an arbitrary graph. In *28th IEEE IPDPS Workshop on Advances in Parallel and Distributed Computational Models*, 2014.

[7] Y. Ding, J.Z. Wang, and P.K. Srimani. Self-stabilizing algorithm for minimal dominating set with safe convergence in an arbitrary graph. *Parallel Processing Letters*, 2014.

[8] Y. Ding, J.Z. Wang, and P.K. Srimani. Self-stabilizing algorithms for maximal 2-packing and general k-packing ($k >= 2$) with safe convergence in an arbitrary graph. *International Journal of Networking and Computing*, 2014.

[9] Y. Ding, J.Z. Wang, and P.K. Srimani. Self-stabilizing master-slave token circulation algorithm in undirected rings and unicyclic graphs of arbitrary size and their orientations. *International Journal of Networking and Computing*, 4(1):42–52, 2014.

[10] Y. Ding, J.Z. Wang, and P.K. Srimani. Self-stabilizing minimal global offensive alliance algorithm with safe convergence in an arbitrary graph. In *11th Annual Conference on Theory and Applications of Models of Computation*, April 2014.

[11] Y. Ding, J.Z. Wang, and P.K. Srimani. Self-stabilizing selection of influential users in social networks. In *13th International Symposium on Pervasive Systems, Algorithms, and Networks*, 2014.

[12] L. Li, Q. Zhang, Y. Ding, H. Jiang, B. Thiers, and J. Wang. A computer-aided spectroscopic system for early diagnosis of melanoma. In *IEEE International Conference on Tools with Artificial Intelligence*, pages 20–24, 2013.